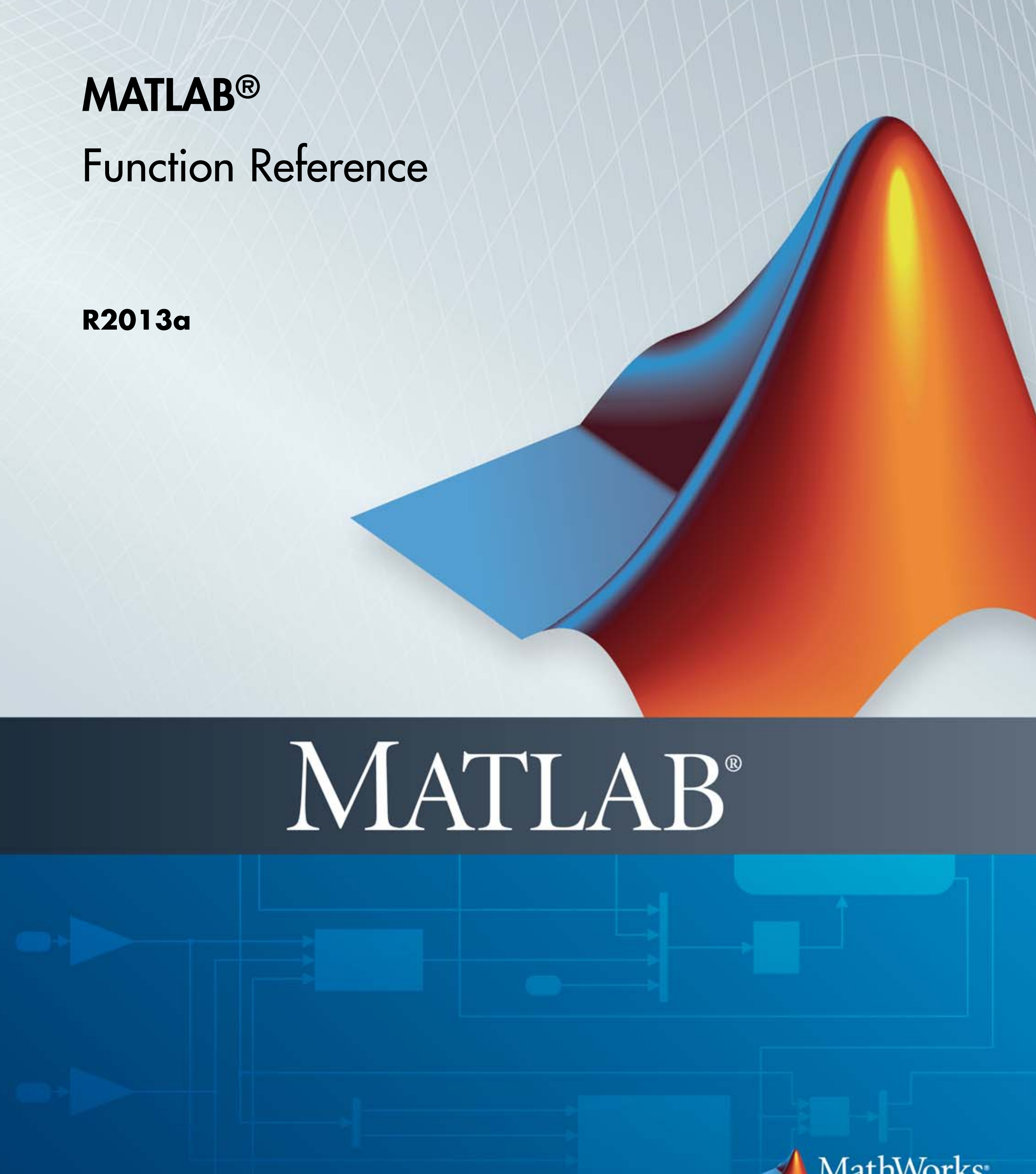
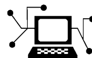

MATLAB®
Function Reference
R2013a
MATLAB®
MathWorks®

How to Contact MathWorks



`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB Function Reference

© COPYRIGHT 1984–2013 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See `www.mathworks.com/trademarks` for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` for more information.

Revision History

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

Contents

1 Alphabetical List

Index

Alphabetical List

Arithmetic Operators + - * / \ ^ ’
Relational Operators < > <= >= == ~=
Logical Operators: Elementwise & | ~
Logical Operators: Short-circuit && ||
Special Characters [ ] ( ) {} = ’ . ... , ; : % ! @
colon (:)
abs
accumarray
acos
acosd
acosh
acot
acotd
acoth
acsc
acscd
acsch
actxcontrol
actxcontrollist
actxcontrolselect
actxGetRunningServer
actxserver
matlab.apputil.create
matlab.apputil.getInstalledAppInfo
matlab.apputil.install
matlab.apputil.package
matlab.apputil.run
matlab.apputil.uninstall

addCause (MException)
addevent
audioinfo
audioread
audiowrite
addframe (avifile)
addlistener (handle)
addOptional
addParamValue
addpath
addpref
addprop (dynamicprops)
addproperty
addRequired
addsampletocollection
addtodate
addts
airy
align
alim
all
allchild
alpha
alphamap
amd
ancestor
and
angle
annotation
Annotation Arrow Properties
Annotation Doublearrow Properties
Annotation Ellipse Properties
Annotation Line Properties
Annotation Rectangle Properties
Annotation Textarrow Properties
Annotation Textbox Properties
ans

any
area
Areaseries Properties
arrayfun
ascii
asec
asecd
asech
asin
asind
asinh
assert
assignin
atan
atan2
atan2d
atand
atanh
audiodevinfo
audioplayer
audiorecorder
aufinfo
auread
auwrite
avifile
aviinfo
aviread
axes
Axes Properties
axis
balance
bar
barh
bar3
bar3h
Barseries Properties
baryToCart

base2dec
beep
BeginInvoke
bench
besselh
besseli
besselj
besselk
bessely
beta
betainc
betaincinv
betaln
bicg
bicgstab
bicgstabl
bin2dec
binary
bitand
bitcmp
bitget
bitmax
bitnot
bitor
bitset
bitshift
bitxor
blanks
blkdiag
box
break
brighten
brush
bsxfun
builddocsearchdb
builtin
bvp4c

bvp5c
bvpget
bvpinit
bvpset
bvpxtend
calendar
calllib
callSoapService
camdolly
cameratoolbar
camlight
camlookat
camorbit
campan
campos
camproj
camroll
camtarget
camup
camva
camzoom
cartToBary
cart2pol
cart2sph
case
cast
cat
catch
caxis
cd
convexHull
cd
cdf2rdf
cdfepoch
cdfinfo
cdflib
cdflib.close

cdflib.closeVar
cdflib.computeEpoch
cdflib.computeEpoch16
cdflib.create
cdflib.createAttr
cdflib.createVar
cdflib.delete
cdflib.deleteAttr
cdflib.deleteAttrEntry
cdflib.deleteAttrgEntry
cdflib.deleteVar
cdflib.deleteVarRecords
cdflib.epoch16Breakdown
cdflib.epochBreakdown
cdflib.getAttrEntry
cdflib.getAttrgEntry
cdflib.getAttrMaxEntry
cdflib.getAttrMaxgEntry
cdflib.getAttrName
cdflib.getAttrNum
cdflib.getAttrScope
cdflib.getCacheSize
cdflib.getChecksum
cdflib.getCompression
cdflib.getCompressionCacheSize
cdflib.getConstantNames
cdflib.getConstantValue
cdflib.getCopyright
cdflib.getFileBackward
cdflib.getFormat
cdflib.getLibraryCopyright
cdflib.getLibraryVersion
cdflib.getMajority
cdflib.getName
cdflib.getNumAttrEntries
cdflib.getNumAttrgEntries
cdflib.getNumAttributes

cdflib.getNumgAttributes
cdflib.getReadOnlyMode
cdflib.getStageCacheSize
cdflib.getValidate
cdflib.getVarAllocRecords
cdflib.getVarBlockingFactor
cdflib.getVarCacheSize
cdflib.getVarCompression
cdflib.getVarData
cdflib.getVarMaxAllocRecNum
cdflib.getVarMaxWrittenRecNum
cdflib.getVarsMaxWrittenRecNum
cdflib.getVarName
cdflib.getVarNum
cdflib.getVarNumRecsWritten
cdflib.getVarPadValue
cdflib.getVarRecordData
cdflib.getVarReservePercent
cdflib.getVarSparseRecords
cdflib.getVersion
cdflib.hyperGetVarData
cdflib.hyperPutVarData
cdflib.inquire
cdflib.inquireAttr
cdflib.inquireAttrEntry
cdflib.inquireAttrgEntry
cdflib.inquireVar
cdflib.open
cdflib.putAttrEntry
cdflib.putAttrgEntry
cdflib.putVarData
cdflib.putVarRecordData
cdflib.renameAttr
cdflib.renameVar
cdflib.setCacheSize
cdflib.setChecksum
cdflib.setCompression

cdflib.setCompressionCacheSize
cdflib.setFileBackward
cdflib.setFormat
cdflib.setMajority
cdflib.setReadOnlyMode
cdflib.setStageCacheSize
cdflib.setValidate
cdflib.setVarAllocBlockRecords
cdflib.setVarBlockingFactor
cdflib.setVarCacheSize
cdflib.setVarCompression
cdflib.setVarInitialRecs
cdflib.setVarPadValue
cdflib.SetVarReservePercent
cdflib.setVarsCacheSize
cdflib.setVarSparseRecords
cdfread
cdfwrite
ceil
cell
cell2mat
cell2struct
celldisp
cellfun
cellplot
cellstr
cgs
char
checkcode
checkin
checkout
chol
cholinc
cholupdate
circshift
circumcenters
cla

clabel
class
classdef
clc
clear
clearvars
clear (serial)
clf
clipboard
clock
close
close
close (avifile)
close
close
closereq
cmopts
cmpermute
cmunique
colamd
colorbar
colordef
colormap
colormapeditor
ColorSpec (Color Specification)
colperm
Combine
comet
comet3
commandhistory
commandwindow
compan
compass
complex
computeStrip
computeTile
computer

cond
condeig
condest
coneplot
conj
continue
contour
contour3
contourc
contourf
Contourgroup Properties
contourslice
contrast
conv
conv2
convhull
convhulln
convn
matlab.mixin.Copyable
copy
copyfile
copyobj
corrcoef
cos
cosd
cosh
cot
cotd
coth
cov
cplxpair
cputime
RandStream.create
createClassFromWsdl
createCopy
createSoapMessage
cross

csc
cscd
csch
csvread
ctranspose
csvwrite
cumprod
cumsum
cumtrapz
curl
currentDirectory
customverctrl
cylinder
daqread
daspect
datacursormode
datatipinfo
date
datenum
datestr
datetick
datevec
dbclear
dbcont
dbdown
dblquad
dbmex
dbquit
dbstack
dbstatus
dbstep
dbstop
dbtype
dbup
dde23
ddeget
ddensd

ddesd
ddeset
deal
deblank
dec2base
dec2bin
dec2hex
decic
deconv
del2
DelaunayTri
DelaunayTri
delaunay
delaunayn
delaunayTriangulation
convexHull
isInterior
nearestNeighbor
pointLocation
voronoiDiagram
delete
delete (COM)
delete
delete (handle)
delete (serial)
delete (timer)
deleteproperty
delevent
delsamplefromcollection
demo
depdir
depfun
det
detrend
deval
diag
dialog

diary
diff
diffuse
dir
dir
disp
disp (memmapfile)
disp (MException)
disp (serial)
disp (timer)
display
dither
divergence
dlmread
dlmwrite
dmperm
doc
docsearch
dos
dot
double
dragrect
drawnow
dsearchn
dynamicprops
echo
echodemo
edgeAttachments
edges
edit
eig
eigs
ellipj
ellipke
ellipsoid
else
elseif

empty
enableNETfromNetworkDrive
enableservice
end
EndInvoke
eomday
enumeration
eps
eq
eq (MException)
erf
erfc
erfcinv
erfcx
erfinv
error
errorbar
Errorbarseries Properties
errordlg
etime
etree
etreeplot
eval
evalc
evalin
event.EventData
event.listener
event.PropertyEvent
event.proplistener
eventlisteners
events
events (COM)
Execute
exifread
exist
exit
exp

expint
expm
expm1
export2wsdlg
eye
ezcontour
ezcontourf
ezmesh
ezmeshc
ezplot
ezplot3
ezpolar
ezsurf
ezsurfc
faceNormals
factor
factorial
false
fclose
fclose (serial)
feather
featureEdges
feof
ferror
feval
Feval (COM)
fft
fft2
fftn
fftshift
fftw
fgetl
fgetl (serial)
fgets
fgets (serial)
fieldnames
figure

Figure Properties
figurepalette
fileattrib
filebrowser
filemarker
fileparts
fileread
filesep
fill
fill3
filter
filter2
find
findall
findfigs
findobj
findobj (handle)
findprop (handle)
findstr
finish
fitsdisp
fitsinfo
fitsread
fitswrite
fix
flintmax
flipdim
fliplr
flipud
floor
flow
fminbnd
fminsearch
fopen
fopen (serial)
for
format

fplot
fprintf
fprintf (serial)
frame2im
fread
fread (serial)
freeBoundary
freqspace
frewind
fscanf
fscanf (serial)
fseek
ftell
FTP
full
fullfile
func2str
function
function_handle (@)
functions
funm
fwrite
fwrite (serial)
fzero
gallery
gamma
gammainc
gammaincinv
gammaln
gca
gcbf
gcbo
gcd
gcf
gco
ge
genpath

genvarname
get
get
get
get (COM)
get (hgsetget)
get (memmapfile)
get
get (RandStream)
get (serial)
get (timer)
get (tscollection)
getabstime (tscollection)
getappdata
getaudiodata
GetCharArray
RandStream.getDefaultStream
getdisp (hgsetget)
getenv
getfield
getFileFormats
getframe
GetFullMatrix
getpixelposition
getpref
getProfiles
getReport (MException)
getsampleusingtime (tscollection)
getTag
getTagNames
gettimeseriesnames
gettsafteratevent
gettsafterevent
gettsatevent
gettsbeforeatevent
gettsbeforeevent
gettsbetweenevents

GetVariable
getVersion
GetWorkspaceData
ginput
global
gmres
gobjects
gplot
grabcode
gradient
graymon
grid
griddata
griddatan
griddedInterpolant
gsvd
gt
gtext
guidata
guide
guihandles
gunzip
gzip
h5create
h5disp
h5info
h5read
h5readatt
h5write
h5writeatt
H5.close
H5.garbage_collect
H5.get_libversion
H5.open
H5.set_free_list_limits
H5A.close
H5A.create

H5A.delete
H5A.get_info
H5A.get_name
H5A.get_space
H5A.get_type
H5A.iterate
H5A.open
H5A.open_by_idx
H5A.open_by_name
H5A.read
H5A.write
H5D.close
H5D.create
H5D.get_access_plist
H5D.get_create_plist
H5D.get_offset
H5D.get_space
H5D.get_space_status
H5D.get_storage_size
H5D.get_type
H5D.open
H5D.read
H5D.set_extent
H5D.vlen_get_buf_size
H5D.write
H5DS.attach_scale
H5DS.detach_scale
H5DS.get_label
H5DS.get_num_scales
H5DS.get_scale_name
H5DS.is_scale
H5DS.iterate_scales
H5DS.set_label
H5DS.set_scale
H5E.clear
H5E.get_major
H5E.get_minor

H5E.walk
H5F.close
H5F.create
H5F.flush
H5F.get_access_plist
H5F.get_create_plist
H5F.get_filesize
H5F.get_freespace
H5F.get_info
H5F.get_mdc_config
H5F.get_mdc_hit_rate
H5F.get_mdc_size
H5F.get_name
H5F.get_obj_count
H5F.get_obj_ids
H5F.is_hdf5
H5F.mount
H5F.open
H5F.reopen
H5F.set_mdc_config
H5F.unmount
H5G.close
H5G.create
H5G.get_info
H5G.open
H5I.dec_ref
H5I.get_file_id
H5I.get_name
H5I.get_ref
H5I.get_type
H5I.inc_ref
H5I.is_valid
H5L.copy
H5L.create_external
H5L.create_hard
H5L.create_soft
H5L.delete

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

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

H5P.set_fletcher32
H5P.set_layout
H5P.set_nbit
H5P.set_scaleoffset
H5P.set_shuffle
H5P.get_alignment
H5P.get_driver
H5P.get_family_offset
H5P.get_fapl_core
H5P.get_fapl_family
H5P.get_fapl_multi
H5P.get_fclose_degree
H5P.get_libver_bounds
H5P.get_gc_references
H5P.get_mdc_config
H5P.get_meta_block_size
H5P.get_multi_type
H5P.get_sieve_buf_size
H5P.get_small_data_block_size
H5P.set_alignment
H5P.set_family_offset
H5P.set_fapl_core
H5P.set_fapl_family
H5P.set_fapl_log
H5P.set_fapl_multi
H5P.set_fapl_sec2
H5P.set_fapl_split
H5P.set_fapl_stdio
H5P.set_fclose_degree
H5P.set_gc_references
H5P.set_libver_bounds
H5P.set_mdc_config
H5P.set_meta_block_size
H5P.set_multi_type
H5P.set_sieve_buf_size
H5P.set_small_data_block_size
H5P.get_istore_k

H5P.get_sizes
H5P.get_sym_k
H5P.get_userblock
H5P.get_version
H5P.set_istore_k
H5P.set_sizes
H5P.set_sym_k
H5P.set_userblock
H5P.get_attr_creation_order
H5P.get_attr_phase_change
H5P.get_copy_object
H5P.set_attr_creation_order
H5P.set_attr_phase_change
H5P.set_copy_object
H5P.get_create_intermediate_group
H5P.get_link_creation_order
H5P.get_link_phase_change
H5P.set_create_intermediate_group
H5P.set_link_creation_order
H5P.set_link_phase_change
H5P.get_char_encoding
H5P.set_char_encoding
H5R.create
H5R.dereference
H5R.get_name
H5R.get_obj_type
H5R.get_region
H5S.copy
H5S.create
H5S.close
H5S.create_simple
H5S.extent_copy
H5S.get_select_bounds
H5S.get_select_elem_npoints
H5S.get_select_elem_pointlist
H5S.get_select_hyper_blocklist
H5S.get_select_hyper_nblocks

H5S.get_select_npoints
H5S.get_select_type
H5S.get_simple_extent_dims
H5S.get_simple_extent_ndims
H5S.get_simple_extent_npoints
H5S.get_simple_extent_type
H5S.is_simple
H5S.offset_simple
H5S.select_all
H5S.select_elements
H5S.select_hyperslab
H5S.select_none
H5S.select_valid
H5S.set_extent_none
H5S.set_extent_simple
H5T.close
H5T.commit
H5T.committed
H5T.copy
H5T.create
H5T.detect_class
H5T.equal
H5T.get_class
H5T.get_create_plist
H5T.get_native_type
H5T.get_size
H5T.get_super
H5T.lock
H5T.open
H5T.array_create
H5T.get_array_dims
H5T.get_array_ndims
H5T.get_cset
H5T.get_ebias
H5T.get_fields
H5T.get_inpad
H5T.get_norm

H5T.get_offset
H5T.get_order
H5T.get_pad
H5T.get_precision
H5T.get_sign
H5T.get_strpad
H5T.set_cset
H5T.set_ebias
H5T.set_fields
H5T.set_inpad
H5T.set_norm
H5T.set_offset
H5T.set_order
H5T.set_pad
H5T.set_precision
H5T.set_sign
H5T.set_size
H5T.set_strpad
H5T.get_member_class
H5T.get_member_index
H5T.get_member_name
H5T.get_member_offset
H5T.get_member_type
H5T.get_nmembers
H5T.insert
H5T.pack
H5T.enum_create
H5T.enum_insert
H5T.enum_nameof
H5T.enum_valueof
H5T.get_member_value
H5T.get_tag
H5T.set_tag
H5T.is_variable_str
H5T.vlen_create
H5Z.filter_avail
H5Z.get_filter_info

hadamard
handle
hankel
hdf
hdf5info
hdf5read
hdf5write
hdfinfo
hdfread
hdftool
help
helpbrowser
helpdesk
helpdlg
helpwin
hess
matlab.mixin.Heterogeneous
cat
getDefaultScalarElement
horzcat
vertcat
hex2dec
hex2num
hgexport
hggroup
Hggroup Properties
hgload
hgsave
hgsetget
hgtransform
Hgtransform Properties
hidden
hilb
hist
histc
hold
home

horzcat
horzcat (tscollection)
hostid
hsv2rgb
hypot
i
ichol
idivide
if/elseif/else
ifft
ifft2
ifftn
ifftshift
ilu
im2frame
im2java
imag
image
Image Properties
imagesc
imapprox
imfinfo
imformats
import
importdata
imread
imwrite
incenters
inOutStatus
ind2rgb
ind2sub
Inf
inferiorto
info
inline
inmem
inpolygon

input
inputdlg
inputname
inputParser
inspect
instrcallback
instrfind
instrfindall
int2str
int8
int16
int32
int64
integral
integral2
integral3
interfaces
interp1
interp1q
interp2
interp3
interpft
interpn
interpstreamspeed
intersect
intmax
intmin
inv
invhilb
invoke
ipermute
is*
isa
isappdata
iscell
iscellstr
ischar

iscolumn
iscom
isdir
isEdge
isempty
isempty (tscollection)
isequal
isequaln
isequal (MException)
isequalwithequalnans
isevent
isfield
isfinite
isfloat
isglobal
ishandle
ishghandle
ishold
isinf
isinteger
isinterface
isjava
isKey
iskeyword
isletter
islogical
ismac
ismatrix
ismember
ismethod
ismethod (COM)
isnan
isnumeric
isobject
isocaps
isocolors
isonormals

isosurface
ispc
ispref
isprime
isprop
isprop (COM)
isreal
isrow
isscalar
issorted
isspace
issparse
isstr
isstrprop
isstruct
isstudent
isTiled
isunix
isvalid (handle)
isvalid (serial)
isvalid (timer)
isvarname
isvector
j
javaaddpath
javaArray
javachk
javaclasspath
matlab.exception.JavaException
javaMethod
javaMethodEDT
javaObject
javaObjectEDT
javarmpath
keyboard
keys
kron

last (MException)
lastDirectory
lasterr
lasterror
lastwarn
lcm
ldl
ldivide
le
legend
legendre
length
length
length (serial)
length (tscollection)
libfunctions
libfunctionsview
libisloaded
libpointer
lib.pointer
disp
isnull
plus
reshape
setdatatype
libstruct
license
light
Light Properties
lightangle
lighting
lin2mu
line
Line Properties
Lineseries Properties
LineSpec (Line Specification)
linkaxes

linkdata
linkprop
linsolve
linspace
RandStream.list
listdlg
listfonts
load
load (COM)
load (serial)
loadlibrary
loadobj
log
log10
log1p
log2
logical
loglog
logm
logspace
lookfor
lower
ls
lscov
lsqnonneg
lsqr
lt
lu
luinc
magic
makehgtform
containers.Map
mat2cell
mat2str
material
matfile
matlab.io.MatFile

matlabrc
matlabroot
matlab (UNIX)
matlab (Windows)
max
MaximizeCommandWindow
maxNumCompThreads
mean
median
memmapfile
memory
menu
mesh
meshc
meshz
meshgrid
meta.class
meta.class.fromName
meta.DynamicProperty
meta.EnumeratedValue
meta.event
meta.MetaData
meta.method
meta.package
meta.abstractDetails
meta.package.fromName
meta.package.getAllPackages
meta.property
metaclass
methods
methodsview
mex
mex.getCompilerConfigurations
MException
mexext
mfilename
mget

min
MinimizeCommandWindow
minres
minus
mislocked
mkdir
mkdir
mkpp
mldivide
mrdivide
mlint
mlintrpt
mlock
mmfileinfo
mmreader
mod
mode
more
move
movefile
movegui
movie
movie2avi
mpower
mput
msgbox
mtimes
mu2lin
multibandread
multibandwrite
munlock
namelengthmax
NaN
nargchk
nargin
narginchk
nargout

nargoutchk
native2unicode
nchoosek
ndgrid
ndims
ne
nearestNeighbor
ne (MException)
neighbors
NET
NET.addAssembly
NET.Assembly
NET.convertArray
NET.createArray
NET.createGeneric
NET.disableAutoRelease
NET.enableAutoRelease
NET.GenericClass
NET.invokeGenericMethod
NET.isNETSupported
NET.NetException
NET.setStaticProperty
nccreate
ncdisp
ncinfo
ncread
ncreadatt
ncwrite
ncwriteatt
ncwriteschema
netcdf
netcdf.abort
netcdf.close
netcdf.copyAtt
netcdf.create
netcdf.defDim
netcdf.defGrp

netcdf.defVar
netcdf.defVarChunking
netcdf.defVarDeflate
netcdf.defVarFill
netcdf.defVarFletcher32
netcdf.delAtt
netcdf.endDef
netcdf.getAtt
netcdf.getChunkCache
netcdf.getConstant
netcdf.getConstantNames
netcdf.getVar
netcdf.inq
netcdf.inqDimIDs
netcdf.inqFormat
netcdf.inqGrpName
netcdf.inqGrpNameFull
netcdf.inqGrpParent
netcdf.inqGrps
netcdf.inqNcid
netcdf.inqUnlimDims
netcdf.inqVarIDs
netcdf.inqVarChunking
netcdf.inqVarDeflate
netcdf.inqVarFill
netcdf.inqVarFletcher32
netcdf.inqAtt
netcdf.inqAttID
netcdf.inqAttName
netcdf.inqDim
netcdf.inqDimID
netcdf.inqLibVers
netcdf.inqVar
netcdf.inqVarID
netcdf.open
netcdf.putAtt
netcdf.putVar

netcdf.reDef
netcdf.renameAtt
netcdf.renameDim
netcdf.renameVar
netcdf.setChunkCache
netcdf.setDefaultFormat
netcdf.setFill
netcdf.sync
newplot
nextDirectory
nextpow2
nnz
noanimate
nonzeros
norm
normest
not
notebook
notify (handle)
now
nthroot
null
num2cell
num2hex
num2str
numberOfStrips
numberOfTiles
numel
nzmax
ode15i
ode15s
ode23
ode23s
ode23t
ode23tb
ode45
ode113

odeget
odeset
odextend
onCleanup
ones
open
open
openfig
opengl
openvar
optimget
optimset
or
ordeig
orderfields
ordqz
ordschur
orient
orth
otherwise
pack
padecoef
pagesetupdlg
pan
pareto
parfor
parse
parseSoapResponse
pascal
patch
Patch Properties
path
path2rc
pathsep
pathtool
pause
pbaspect

pcg
pchip
pcode
pcolor
pdepe
pdeval
peaks
perl
perms
permute
persistent
pi
pie
pie3
pinv
planerot
play
play
playblocking
plot
plot3
plotbrowser
plotedit
plotmatrix
plottools
plotyy
plus
pointLocation
pol2cart
polar
poly
polyarea
polyder
polyeig
polyfit
polyint
polyval

polyvalm
pow2
power
ppval
prefdir
preferences
primes
print
printopt
printdlg
printpreview
prod
profile
profsave
propedit
propedit (COM)
properties
propertyeditor
psi
publish
PutCharArray
PutFullMatrix
PutWorkspaceData
pwd
qmr
qr
qrdelete
qrinsert
qrupdate
quad
quad2d
quadgk
quadl
quadv
questdlg
quit
Quit (COM)

quiver
quiver3
Quivergroup Properties
qz
rand
rand (RandStream)
randi
randi (RandStream)
randn
randn (RandStream)
randperm
randperm (RandStream)
RandStream
RandStream constructor
RandStream.getGlobalStream
RandStream.setGlobalStream
rank
rat
rats
rbbox
rcond
rdivide
read
read
readasync
readEncodedStrip
readEncodedTile
readRGBAImage
readRGBAStrip
readRGBATile
Remove
RemoveAll
timeseries
addsample
append
ctranspose
delsample

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

isConnected
neighbors
size
vertexAttachments
vertexNormal
real
reallog
realmax
realmin
realpow
realsqrt
record
record
recordblocking
rectangle
Rectangle Properties
rectint
recycle
reducepatch
reducevolume
refresh
refreshdata
regexp
regexpi
regexprep
regexptranslate
registerevent
rehash
release
relationaloperators (handle)
rem
remove
removets
rename
repmat
resample (tscollection)
reset

reset (RandStream)
reshape
residue
restoredefaultpath
rethrow
rethrow (MException)
return
rewriteDirectory
rgb2hsv
rgb2ind
rgbplot
ribbon
rmappdata
rmdir
rmdir
rmfield
rmpath
rmpref
rng
root object
Root Properties
roots
rose
rosser
rot90
rotate
rotate3d
round
rref
rsf2csf
run
save
save (COM)
save (serial)
saveas
saveobj
savepath

scatter
scatter3
Scattergroup Properties
schur
script
scatteredInterpolant
sec
secd
sech
selectmoveresize
semilogx
semilogy
sendmail
serial
serialbreak
set
set
set
set (COM)
set (hgsetget)
set
set (RandStream)
set (serial)
set (timer)
set (tscollection)
setabstime (tscollection)
setappdata
RandStream.setDefaultStream
setdiff
setDirectory
setdisp (hgsetget)
setenv
setfield
setpixelposition
setpref
setstr
setSubDirectory

setTag
settimeseriesnames
setxor
shading
shg
shiftdim
showplottool
shrinkfaces
sign
sin
sind
single
sinh
size
size
size
size (serial)
size
size (tscollection)
slice
smooth3
snapnow
sort
sortrows
sound
soundsc
spalloc
sparse
spaugment
spconvert
spdiags
specular
speye
spfun
sph2cart
sphere
spinmap

spline
spones
spparms
sprand
sprandn
sprandsym
sprank
sprintf
spy
sqrt
sqrtm
squeeze
ss2tf
sscanf
stairs
Stairseries Properties
start
startat
startup
std
stem
stem3
Stemseries Properties
stop
stopasync
str2double
str2func
str2mat
str2num
strcat
strcmp
strcmpi
stream2
stream3
streamline
streamparticles
streamribbon

streamslice
streamtube
strfind
strings
strjoin
strjust
strmatch
strncmp
strncmpi
strread
strrep
strsplit
strtok
strtrim
struct
struct2cell
structfun
strvcat
sub2ind
subplot
subsasgn
subsindex
subspace
subsref
substruct
subvolume
sum
superclasses
superiorto
support
surf
surfc
surf2patch
surface
Surface Properties
Surfaceplot Properties
surfl

surfnorm
svd
svds
swapbytes
switch/case/otherwise
symamd
symbfact
symmlq
symrcm
symvar
syntax
system
tan
tand
tanh
tar
tempdir
tempname
tetramesh
matlab.unittest.Test
matlab.unittest.TestCase
addTeardown
assertClass
assertEmpty
assertEqual
assertError
assertFail
assertFalse
assertGreaterThan
assertGreaterThanOrEqual
assertInstanceOf
assertLength
assertLessThan
assertLessThanOrEqual
assertMatches
assertNotEmpty
assertNotEqual

assertNotSameHandle
assertNumElements
assertReturnsTrue
assertSameHandle
assertSize
assertSubstring
assertThat
assertTrue
assertWarning
assertWarningFree
assumeClass
assumeEmpty
assumeEqual
assumeError
assumeFail
assumeFalse
assumeGreaterThan
assumeGreaterThanOrEqual
assumeInstanceOf
assumeLength
assumeLessThan
assumeLessThanOrEqual
assumeMatches
assumeNotEmpty
assumeNotEqual
assumeNotSameHandle
assumeNumElements
assumeReturnsTrue
assumeSameHandle
assumeSize
assumeSubstring
assumeThat
assumeTrue
assumeWarning
assumeWarningFree
fatalAssertClass
fatalAssertEmpty

fatalAssertEqual
fatalAssertError
fatalAssertFail
fatalAssertFalse
fatalAssertGreaterThan
fatalAssertGreaterThanOrEqual
fatalAssertInstanceOf
fatalAssertLength
fatalAssertLessThan
fatalAssertLessThanOrEqual
fatalAssertMatches
fatalAssertNotEmpty
fatalAssertNotEqual
fatalAssertNotSameHandle
fatalAssertNumElements
fatalAssertReturnsTrue
fatalAssertSameHandle
fatalAssertSize
fatalAssertSubstring
fatalAssertThat
fatalAssertTrue
fatalAssertWarning
fatalAssertWarningFree
run
verifyClass
verifyEmpty
verifyEqual
verifyError
verifyFail
verifyFalse
verifyGreaterThan
verifyGreaterThanOrEqual
verifyInstanceOf
verifyLength
verifyLessThan
verifyLessThanOrEqual
verifyMatches

verifyNotEmpty
verifyNotEqual
verifyNotSameHandle
verifyNumElements
verifyReturnsTrue
verifySameHandle
verifySize
verifySubstring
verifyThat
verifyTrue
verifyWarning
verifyWarningFree
matlab.unittest.TestResult
matlab.unittest.TestRunner
addPlugin
run
withNoPlugins
withTextOutput
matlab.unittest.TestSuite
fromClass
fromFile
fromFolder
fromMethod
fromPackage
run
texlabel
text
Text Properties
textread
textscan
textwrap
tfqmr
throw (MException)
throwAsCaller (MException)
tic
toc
Tiff

timer
timerfind
timerfindall
times
title
todatenum
toeplitz
toolboxdir
trace
transpose
trapz
treelayout
treeplot
tril
trimesh
triplequad
triplot
TriRep
TriRep
TriScatteredInterp
TriScatteredInterp
trisurf
triu
true
try/catch
tscollection
tsdata.event
tsearchn
tstool
type
typecast
uibuttongroup
Uibuttongroup Properties
uicontextmenu
Uicontextmenu Properties
uicontrol
Uicontrol Properties

uigetdir
uigetfile
uigetpref
uiimport
uimenu
Uimenu Properties
uint8
uint16
uint32
uint64
uiopen
uipanel
Uipanel Properties
uipushtool
Uipushtool Properties
uiputfile
uiresume
uisave
uisetcolor
uisetfont
uisetpref
uistack
uitable
Uitable Properties
uitoggletool
Uitoggletool Properties
uitoolbar
Uitoolbar Properties
uiwait
uminus
undocheckout
unicode2native
union
unique
unix
unloadlibrary
unmesh

unmkpp
unregisterallevents
unregisterevent
untar
unwrap
unzip
uplus
upper
urlread
urlwrite
usejava
userpath
validateattributes
validatestring
values
vander
var
varargin
varargout
vectorize
ver
verctrl
verLessThan
version
vertcat
vertcat (tscollection)
vertexAttachments
VideoReader
VideoWriter
view
viewmtx
visdiff
volumebounds
voronoi
voronoiDiagram
voronoin
wait

waitbar
waitfor
waitforbuttonpress
warndlg
warning
waterfall
wavfinfo
wavplay
wavread
wavrecord
wavwrite
web
weekday
what
whatsnew
which
while
whitebg
who
who
whos
whos
wilkinson
winopen
winqueryreg
wk1finfo
wk1read
wk1write
workspace
write
writeDirectory
writeEncodedStrip
writeEncodedTile
writeVideo
xlabel
ylabel
zlabel

xlim
ylim
zlim
xlsfinfo
xlsread
xlswrite
xmlread
xmlwrite
xor
xslt
zeros
zip
zoom

Arithmetic Operators + - * / \ ^ ’

Purpose

Matrix and array arithmetic

Syntax

```
A+B
A-B
A*B
A.*B
A/B
A./B
A\B
A.\B
A^B
A.^B
A'
A.'
```

Description

MATLAB® software has two different types of arithmetic operations. Matrix arithmetic operations are defined by the rules of linear algebra. Array arithmetic operations are carried out element by element, and can be used with multidimensional arrays. The period character (`.`) distinguishes the array operations from the matrix operations. However, since the matrix and array operations are the same for addition and subtraction, the character pairs `.+` and `.-` are not used.

+	Addition or unary plus. `A+B` adds `A` and `B`. `A` and `B` must have the same size, unless one is a scalar. A scalar can be added to a matrix of any size.
-	Subtraction or unary minus. `A-B` subtracts `B` from `A`. `A` and `B` must have the same size, unless one is a scalar. A scalar can be subtracted from a matrix of any size.

`*`	Matrix multiplication. `C = A*B` is the linear algebraic product of the matrices `A` and `B`. More precisely, $$C(i,j) = \sum_{k=1}^{n} A(i,k)B(k,j)$$ For nonscalar `A` and `B`, the number of columns of `A` must equal the number of rows of `B`. A scalar can multiply a matrix of any size.
`.*`	Array multiplication. `A.*B` is the element-by-element product of the arrays `A` and `B`. `A` and `B` must have the same size, unless one of them is a scalar.
`/`	Slash or matrix right division. `B/A` is roughly the same as `B*inv(A)`. More precisely, `B/A = (A'\B')'`. See the reference page for `mrdivide` for more information.
`./`	Array right division. `A./B` is the matrix with elements `A(i,j)/B(i,j)`. `A` and `B` must have the same size, unless one of them is a scalar.
`\`	Backslash or matrix left division. If `A` is a square matrix, `A\B` is roughly the same as `inv(A)*B`, except it is computed in a different way. If `A` is an `n`-by-`n` matrix and `B` is a column vector with `n` components, or a matrix with several such columns, then `X = A\B` is the solution to the equation $AX = B$. A warning message is displayed if `A` is badly scaled or nearly singular. See the reference page for `mldivide` for more information.

If `A` is an `m`-by-`n` matrix with `m ~= n` and `B` is a column vector with `m` components, or a matrix with several such columns, then `X = A\B` is the solution in the least squares sense to the under- or overdetermined system of equations $AX = B$. The effective rank, `k`, of `A` is determined from the QR decomposition with pivoting. A solution `X` is computed that has at most `k` nonzero components per column. If `k < n`, this is usually not the same solution as `pinv(A)*B`, which is the least squares solution with the smallest norm $\|X\|$.

`.\` Array left division. `A.\B` is the matrix with elements `B(i,j)/A(i,j)`. `A` and `B` must have the same size, unless one of them is a scalar.

`^` Matrix power. `X^p` is `X` to the power `p`, if `p` is a scalar. If `p` is an integer, the power is computed by repeated squaring. If the integer is negative, `X` is inverted first. For other values of `p`, the calculation involves eigenvalues and eigenvectors, such that if `[V,D] = eig(X)`, then `X^p = V*D.^p/V`.

If `x` is a scalar and `P` is a matrix, `x^P` is `x` raised to the matrix power `P` using eigenvalues and eigenvectors. `X^P`, where `X` and `P` are both matrices, is an error.

`.^` Array power. `A.^B` is the matrix with elements `A(i,j)` to the `B(i,j)` power. `A` and `B` must have the same size, unless one of them is a scalar.

`'` Matrix transpose. `A'` is the linear algebraic transpose of `A`. For complex matrices, this is the complex conjugate transpose.

`.'` Array transpose. `A.'` is the array transpose of `A`. For complex matrices, this does not involve conjugation.

Nondouble Data Type Support

This section describes the arithmetic operators' support for data types other than `double`.

Data Type single

You can apply any of the arithmetic operators to arrays of type `single` and MATLAB software returns an answer of type `single`. You can also combine an array of type `double` with an array of type `single`, and the result has type `single`.

Integer Data Types

You can apply most of the arithmetic operators to real arrays of the following integer data types:

- `int8` and `uint8`
- `int16` and `uint16`
- `int32` and `uint32`
- `int64` and `uint64`

All operands must have the same integer data type and MATLAB returns an answer of that type.

Note Except for the unary operators `+A` and `A.'`, the arithmetic operators do not support operations on complex arrays of any integer data type.

For example,

```
x = int8(3) + int8(4);
class(x)

ans =

int8
```

The following table lists the binary arithmetic operators that you can apply to arrays of the same integer data type. In the table, `A` and `B` are arrays of the same integer data type and `c` is a scalar of type `double` or the same type as `A` and `B`.

Operation	Support when A and B Have Same Integer Type
`+A, -A`	Yes
`A+B, A+c, c+B`	Yes
`A-B, A-c, c-B`	Yes
`A.*B`	Yes
`A*c, c*B`	Yes
`A*B`	No
`A/c, c/B`	Yes
`A.\B, A./B`	Yes
`A\B, A/B`	No
`A.^B`	Yes, if `B` has nonnegative integer values.
`c^k`	Yes, for a scalar `c` and a nonnegative scalar integer `k`, which have the same integer data type or one of which has type `double`
`A.', A'`	Yes

Combining Integer Data Types with Type Double

For the operations that support integer data types, you can combine a scalar or array of an integer data type with a scalar, but not an array, of type `double` and the result has the same integer data type as the input of integer type. For example,

```
y = 5 + int32(7);
class(y)
```

```
ans =

int32
```

However, you cannot combine an array of an integer data type with either of the following:

- A scalar or array of a different integer data type
- A scalar or array of type `single`

Tips

The arithmetic operators have function equivalents, as shown here:

Binary addition	`A+B`	`plus(A,B)`
Unary plus	`+A`	`uplus(A)`
Binary subtraction	`A-B`	`minus(A,B)`
Unary minus	`-A`	`uminus(A)`
Matrix multiplication	`A*B`	`mtimes(A,B)`
Arraywise multiplication	`A.*B`	`times(A,B)`
Matrix right division	`A/B`	`mrdivide(A,B)`
Arraywise right division	`A./B`	`rdivide(A,B)`
Matrix left division	`A\B`	`mldivide(A,B)`
Arraywise left division	`A.\B`	`ldivide(A,B)`
Matrix power	`A^B`	`mpower(A,B)`
Arraywise power	`A.^B`	`power(A,B)`

Complex transpose	`A'`	`ctranspose(A)`
Matrix transpose	`A.'`	`transpose(A)`

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

Examples

Here are two vectors, and the results of various matrix and array operations on them, printed with `format rat`.

Matrix Operations		Array Operations	
x	1 2 3	y	4 5 6
x'	1 2 3	y'	4 5 6
x+y	5 7 9	x-y	-3 -3 -3
x + 2	3 4 5	x-2	-1 0 1
x * y	Error	x.*y	4 10 18

Matrix Operations		Array Operations	
x'*y	32	x'.*y	Error
x*y'	4 5 6 8 10 12 12 15 18	x.*y'	Error
x*2	2 4 6	x.*2	2 4 6
x\y	16/7	x.\y	4 5/2 2
2\x	1/2 1 3/2	2./x	2 1 2/3
x/y	0 0 1/6 0 0 1/3 0 0 1/2	x./y	1/4 2/5 1/2
x/2	1/2 1 3/2	x./2	1/2 1 3/2
x^y	Error	x.^y	1 32 729

Matrix Operations		Array Operations	
`x^2`	Error	`x.^2`	1 4 9
`2^x`	Error	`2.^x`	2 4 8
`(x+i*y)'`	1 - 4i 2 - 5i 3 - 6i		
`(x+i*y).'`	1 + 4i 2 + 5i 3 + 6i		

Diagnostics

- From matrix division, if a square A is singular,

```
Warning: Matrix is singular to working precision.
```

- If the inverse was found, but is not reliable,

```
Warning: Matrix is close to singular or badly scaled.
    Results may be inaccurate.  RCOND = xxx
```

- From matrix division, if a nonsquare A is rank deficient,

```
Warning: Rank deficient, rank = xxx tol = xxx
```

See Also

`mldivide` | `mrdivide` | `chol` | `det` | `inv` | `lu` | `orth` | `permute` | `ipermute` | `qr`

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

Purpose

Relational operations

Syntax

```
A < B
A > B
A <= B
A >= B
A == B
A ~= B
```

Description

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

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

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

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

Examples

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

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

produce the same result:

```
ans =

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

See Also

all | any | find | strcmp | Logical Operators: Elementwise & | ~ | Logical Operators: Short-circuit && ||

Purpose

Elementwise logical operations on arrays

Syntax

```
expr1 & expr2
expr1 | expr2
~expr
```

Description

The symbols `&`, `|`, and `~` are the logical array operators `AND`, `OR`, and `NOT`. These operators are commonly used in conditional statements, such as `if` and `while`, to determine whether or not to execute a particular block of code. Logical operations return a `logical` array with elements set to `1` (`true`) or `0` (`false`), as appropriate.

`expr1 & expr2` represents a logical `AND` operation between values, arrays, or expressions `expr1` and `expr2`. In an AND operation, if `expr1` is `true` *and* `expr2` is `true`, then the AND of those inputs is `true`. If either expression is `false`, the result is `false`. Here is a pseudocode example of AND:

```
IF (expr1: all required inputs were passed) AND ...
   (expr2: all inputs are valid)
THEN (result: execute the function)
```

`expr1 | expr2` represents a logical `OR` operation between values, arrays, or expressions `expr1` and `expr2`. In an OR operation, if `expr1` is `true` *or* `expr2` is true, then the OR of those inputs is `true`. If both expressions are `false`, the result is `false`. Here is a pseudocode example of OR:

```
IF (expr1: S is a string) OR ...
   (expr2: S is a cell array of strings)
THEN (result: parse string S)
```

`~expr` represents a logical `NOT` operation applied to expression `expr`. In a NOT operation, if `expr` is `false`, then the result of the operation is `true`. If `expr` is `true`, the result is `false`. Here is a pseudocode example of NOT:

```
IF (expr: function returned a Success status) is NOT true
```

```
THEN (result: throw an error)
```

The function `xor(A,B)` implements the exclusive `OR` operation.

Logical Operations on Arrays

The expression operands for AND, OR, and NOT are often arrays of nonsingleton dimensions. When this is the case, The MATLAB software performs the logical operation on each element of the arrays. The output is an array that is the same size as the input array or arrays.

If just one operand is an array and the other a scalar, then the scalar is matched against each element of the array. When the operands include two or more nonscalar arrays, the sizes of those arrays must be equal.

This table shows the output of AND, OR, and NOT statements that use scalar and/or array inputs. In the table, `S` is a scalar array, `A` is a nonscalar array, and `R` is the resulting array:

Operation	Result
`S1 & S2`	`R = S1 & S2`
`S & A`	`R(1) = S & A(1); ...` `R(2) = S & A(2); ...`
`A1 & A2`	`R(1) = A1(1) & A2(1);` `R(2) = A1(2) & A2(2); ...`
`S1 | S2`	`R = S1 | S2`
`S | A`	`R(1) = S | A(1);` `R(2) = S | A(2); ...`
`A1 | A2`	`R(1) = A1(1) | A2(1);` `R(2) = A1(2) | A2(2); ...`
`~S`	`R = ~S`
`~A`	`R(1) = ~A(1);` `R(2) = ~A(2), ...`

Compound Logical Statements

The number of expressions that you can evaluate with AND or OR is not limited to two (e.g., A & B). Statements such as the following are also valid:

```
expr1 & expr2 & expr3 | expr4 & expr5
```

Use parentheses to establish the order in which MATLAB evaluates a compound operation. Note the difference in the following two statements:

```
(expr1 & expr2) | (expr3 & expr4)   % 2-component OR
 expr1 & (expr2 | expr3) & expr4    % 3-component AND
```

Operator Precedence

The precedence for the logical operators with respect to each other is shown in the table below. MATLAB always gives the & operator precedence over the | operator. Although MATLAB typically evaluates expressions from left to right, the expression a|b&c is evaluated as a|(b&c). It is a good idea to use parentheses to explicitly specify the intended precedence of statements containing combinations of & and |.

Operator	Operation	Priority
~	NOT	Highest
&	Elementwise AND	
		Elementwise OR	
&&	Short-circuit AND	
||	Short-circuit OR	Lowest

Short-Circuiting in Elementwise Operators

The &, and |operators do not short-circuit. See the documentation on the && and || operators if you need short-circuiting capability.

When used in the context of an if or while expression, and only in this context, the elementwise & **and** | operators use short-circuiting in

evaluating their expressions. That is, A&B and A|B ignore the second operand, B, if the first operand, A, is sufficient to determine the result.

So, although the statement 1|[] evaluates to false, the same statement evaluates to true when used in either an if or while expression:

```
A = 1;   B = [];
if(A|B) disp 'The statement is true',  end;
   The statement is true
```

while the reverse logical expression, which does not short-circuit, evaluates to false:

```
if(B|A) disp 'The statement is true',  end;
```

Another example of short-circuiting with elementwise operators shows that a logical expression such as the one shown below, which under most circumstances is invalid due to a size mismatch between A and B, works within the context of an if or while expression:

The A|B statement generates an error:

```
A = [1 1];   B = [2 0 1];
A|B

Error using  |
Inputs must have the same size.
```

But the same statement used to test an if condition does not error:

```
if (A|B) disp 'The statement is true',  end;
   The statement is true
```

Operator Truth Table

The following is a truth table for the operators and functions in the previous example.

Inputs		and	or	not	xor
A	B	A & B	A | B	~A	xor(A,B)
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	0	0

Equivalent Functions

These logical operators have function equivalents, as shown here.

Logical Operation	Equivalent Function
A & B	and(A,B)
A | B	or(A,B)
~A	not(A)

Examples

Example 1 — Conditional Statement with OR

Using OR in a conditional statement, call function `parseString` on `S`, but only if `S` is a character array or a cell array of strings:

```
if ischar(S) | iscellstr(S)
    parseString(S)
end
```

Example 2 — Array AND Array

Find those elements of array `R` that are both greater than 0.3 AND less then 0.9:

```
rng(0,'twister');
R=rand(5,7)
R =
    0.8147    0.0975    0.1576    0.1419    0.6557    0.7577    0.7060
```

```
    0.9058    0.2785    0.9706    0.4218    0.0357    0.7431    0.0318
    0.1270    0.5469    0.9572    0.9157    0.8491    0.3922    0.2769
    0.9134    0.9575    0.4854    0.7922    0.9340    0.6555    0.0462
    0.6324    0.9649    0.8003    0.9595    0.6787    0.1712    0.0971
(R > 0.3) & (R < 0.9)
ans =
     1     0     0     0     1     1     1
     0     0     0     1     0     1     0
     0     1     0     0     1     1     0
     0     0     1     1     0     1     0
     1     0     1     0     1     0     0
```

Example 3 — Array AND Scalar

Find those elements of array R that are greater than the scalar value of 40:

```
rng(0,'twister');
R = rand(3,5) * 50
R =
   40.7362   45.6688   13.9249   48.2444   47.8583
   45.2896   31.6180   27.3441    7.8807   24.2688
    6.3493    4.8770   47.8753   48.5296   40.0140
R > 40
ans =
     1     1     0     1     1
     1     0     0     0     0
     0     0     1     1     1
```

Example 4 — Check Status with NOT

Throw an error if the return status of a function does NOT indicate success:

```
[Z, status] = myfun(X, Y);
if ~(status == SUCCESS);
   error('Error in function myfun')
end
```

Example 5 — OR of Binary Arrays

This example shows the logical `OR` of the elements in the vector `u` with the corresponding elements in the vector `v`:

```
u = [0 0 1 1 0 1];
v = [0 1 1 0 0 1];
u | v

ans =
   0    1    1    1    0    1
```

See Also

`all` | `any` | `find` | `logical` | `xor` | `true` | `false` | `Logical Operators:  Short-circuit && ||` | `Relational Operators < > <= >= == ~=`

Logical Operators: Short-circuit && ||

Purpose

Logical operations, with short-circuiting capability

Syntax

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

Description

`expr1 && expr2` represents a logical `AND` operation that employs short-circuiting behavior. With short-circuiting, the second operand `expr2` is evaluated only when the result is not fully determined by the first operand `expr1`. For example, if `A` = 0, then the following statement evaluates to `false`, regardless of the value of `B`, so the MATLAB software does not evaluate `B`:

```
A && B
```

These two expressions must each be a valid MATLAB statement that evaluates to a scalar logical result.

`expr1 || expr2` represents a logical `OR` operation that employs short-circuiting behavior.

Note Always use the `&&` and `||` operators when short-circuiting is required. Using the elementwise operators (`&` and `|`) for short-circuiting can yield unexpected results.

Examples

In the following statement, it doesn't make sense to evaluate the relation on the right if the divisor, `b`, is zero. The test on the left is put in to avoid generating a warning under these circumstances:

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

By definition, if any operands of an `AND` expression are `false`, the entire expression must be `false`. So, if `(b ~= 0)` evaluates to `false`, MATLAB assumes the entire expression to be `false` and terminates its evaluation of the expression early. This avoids the warning that would be generated if MATLAB were to evaluate the operand on the right.

See Also

all | any | find | logical | xor | true | false | Logical Operators: Elementwise & | ~ | Relational Operators < > <= >= == ~=

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

Purpose

Special characters

Syntax

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

Description

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

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

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

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

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

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

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

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

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

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

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

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

'	Matrix transpose. `X'` is the complex conjugate transpose of `X`. `X.'` is the nonconjugate transpose.
	Quotation mark. `'any text'` is a vector whose components are the ASCII codes for the characters. A quotation mark within the text is indicated by two quotation marks.
.	Decimal point. `314/100`, `3.14`, and `.314e1` are all the same.
	Element-by-element operations. These are obtained using `.*` , `.^`, `./`, or `.\`. See the Arithmetic Operators page.
.	Field access. `S(m).f` when `S` is a structure, accesses the contents of field `f` of that structure.
.( )	Dynamic Field access. `S.(df)` when `S` is a structure, accesses the contents of dynamic field `df` of that structure. Dynamic field names are defined at runtime.
..	Parent folder. See `cd`.
...	Continuation. Three or more periods at the end of a line continue the current function on the next line. Three or more periods before the end of a line cause the MATLAB software to ignore the remaining text on the current line and continue the function on the next line. This effectively makes a comment out of anything on the current line that follows the three periods. For an example, see "Continue Long Statements on Multiple Lines".
,	Comma. Used to separate matrix subscripts and function arguments. Used to separate statements in multistatement lines. For multistatement lines, the comma can be replaced by a semicolon to suppress printing.
;	Semicolon. Used inside brackets to end rows. Used after an expression or statement to suppress printing or to separate statements.
:	Colon. Create vectors, array subscripting, and `for` loop iterations. See `colon (:)` for details.

`%`	Percent. The percent symbol denotes a comment; it indicates a logical end of line. Any following text is ignored. MATLAB displays the first contiguous comment lines in a function or script file in response to a `help` command.
`%{` `%}`	Percent-brace. The text enclosed within the `%{` and `%}` symbols is a comment block. Use these symbols to insert comments that take up more than a single line in your script of function code. Any text between these two symbols is ignored by MATLAB. With the exception of whitespace characters, the %{ and %} operators must appear alone on the lines that immediately precede and follow the block of help text. Do not include any other text on these lines.
`!`	Exclamation point. Indicates that the rest of the input line is issued as a command to the operating system. See "Running External Commands, Scripts, and Programs" for more information.
`@`	Function handle. MATLAB data type that is a handle to a function. See `function_handle (@)` for details.

Tips

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

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

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

See Also

`Relational Operators < > <= >= == ~=` | `Logical Operators: Elementwise & | ~` | `Arithmetic Operators + - * / \ ^ '`

Purpose

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

Description

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

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

`j:k`	is the same as `[j,j+1,...,k]`, or empty when `j > k`.
`j:i:k`	is the same as `[j,j+i,j+2i, ...,j+m*i]`, where `m = fix((k-j)/i)`, for integer values. For information on the definition of `j:i:k` with floating-point values, see Technical Solution 1-4FLI96. This syntax returns an empty matrix when `i == 0`, `i > 0` and `j > k`, or `i < 0` and `j < k`.

If you specify nonscalar arrays, MATLAB interprets `j:i:k` as `j(1):i(1):k(1)`.

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

`A(:,j)`	is the `j`th column of `A`.
`A(i,:)`	is the `i`th row of `A`.
`A(:,:)`	is the equivalent two-dimensional array. For matrices this is the same as `A`.
`A(j:k)`	is `A(j), A(j+1),...,A(k)`.
`A(:,j:k)`	is `A(:,j), A(:,j+1),...,A(:,k)`.
`A(:,:,k)`	is the `k`th page of three-dimensional array `A`.

`A(i,j,k,:)`	is a vector in four-dimensional array `A`. The vector includes `A(i,j,k,1)`, `A(i,j,k,2)`, `A(i,j,k,3)`, and so on.
`A(:)`	is all the elements of `A`, regarded as a single column. On the left side of an assignment statement, `A(:)` fills `A`, preserving its shape from before. In this case, the right side must contain the same number of elements as `A`.

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

Examples

Using the colon with integers,

```
D = 1:4
```

results in

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

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

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

results in

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

The command

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

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

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

A(:,:,2) =
     1     1     1
     1     2     3
     1     3     6
```

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

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

results in

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

See Also

for | linspace | logspace | reshape | varargin

Purpose

Absolute value and complex magnitude

Syntax

```
abs(X)
```

Description

`abs(X)` returns an array `Y` such that each element of `Y` is the absolute value of the corresponding element of `X`.

If `X` is complex, `abs(X)` returns the complex modulus (magnitude), which is the same as

```
sqrt(real(X).^2 + imag(X).^2)
```

Examples

```
abs(-5)
ans =
     5

abs(3+4i)
ans =
     5
```

See Also

`angle` | `sign` | `unwrap`

Purpose

Construct array with accumulation

Syntax

```
A = accumarray(subs,val)
A = accumarray(subs,val,sz)
A = accumarray(subs,val,sz,fun)
A = accumarray(subs,val,sz,fun,fillval)
A = accumarray(subs,val,sz,fun,fillval,issparse)
A = accumarray({subs1, subs2, ...}, val, ...)
```

Description

`accumarray` groups elements from a data set and applies a function to each group. `A = accumarray(subs,val)` creates an array `A` by accumulating elements of the vector `val` using the elements of `subs` as indices. The position of an element in `subs` determines which value of `vals` it selects for the accumulated vector; the value of an element in `subs` determines the position of the accumulated vector in the output.

`A = accumarray(subs,val,sz)` creates an array `A` with size `sz`, where `sz` is a vector of positive integers. If `subs` is nonempty with N>1 columns, then `sz` must have N elements, where `all(sz >= max(subs,[],1))`. If `subs` is a nonempty column vector, then `sz` must be `[M 1]`, where `M >= MAX(subs)`. Specify `sz` as `[]` for the default behavior.

`A = accumarray(subs,val,sz,fun)` applies function `fun` to each subset of elements of `val`. The default accumulating function is `sum`. To specify another function `fun`, use the `@` symbol (e.g., `@max`). The function `fun` must accept a column vector and return a numeric, logical, or character scalar, or a scalar cell. Return value `A` has the same class as the values returned by `fun`. Specify `fun` as `[]` for the default behavior.

`A = accumarray(subs,val,sz,fun,fillval)` puts the scalar value `fillval` in elements of `A` that are not referred to by any row of `subs`. For example, if `subs` is empty, then `A` is `repmat(fillval,sz)`. `fillval` and the values returned by `fun` must belong to the same class. The default value of `fillval` is 0.

`A = accumarray(subs,val,sz,fun,fillval,issparse)` creates an array `A` that is sparse if the scalar input `issparse` is equal to logical 1 (i.e., `true`), or full if `issparse` is equal to logical 0 (`false`). `A` is full by

default. If `issparse` is `true`, then `fillval` must be zero or `[]`, and `val` and the output of `fun` must be double.

`A = accumarray({subs1, subs2, ...}, val, ...)` passes multiple `subs` vectors in a cell array. You can use any of the four optional inputs (`sz`, `fun`, `fillval`, or `issparse`) with this syntax.

Note If the subscripts in `subs` are not sorted, `fun` should not depend on the order of the values in its input data.

The function processes the input as follows:

1. Find out how many unique indices there are in subs. Each unique index defines a bin in the output array. The maximum index value in `subs` determines the size of the output array.

2. Find out how many times each index is repeated.

   This determines how many elements of `vals` are going to be accumulated at each bin in the output array.

3. Create an output array. The output array is of size `max(subs)` or of size `sz`.

4. Accumulate the entries in `vals` into bins using the values of the indices in `subs` and apply `fun` to the entries in each bin.

5. Fill the values in the output for positions not referred to by `subs`. Default fill value is zero; use `fillval` to set a different value.

Note `subs` should contain positive integers. `subs` can also be a cell vector with one or more elements, each element a vector of positive integers. All the vectors must have the same length. In this case, `subs` is treated as if the vectors formed columns of an index matrix.`val` must be a numeric, logical, or character vector with the same length as the number of rows in `subs`. `val` can also be a scalar whose value is repeated for all the rows of `subs`.

Examples

Example 1

Create a 5-by-1 vector and sum values for repeated 1-D subscripts:

```
val = 101:105;
subs = [1; 2; 4; 2; 4]
subs =
     1
     2
     4
     2
     4

A = accumarray(subs, val)
A =
   101       % A(1) = val(1) = 101
   206       % A(2) = val(2)+val(4) = 102+104 = 206
     0       % A(3) = 0
   208       % A(4) = val(3)+val(5) = 103+105 = 208
```

Example 2

Create a 4-by-4 matrix and subtract values for repeated 2-D subscripts:

```
val = 101:106;
subs=[1 2; 1 2; 3 1; 4 1; 4 4; 4 1];
B = accumarray(subs,val,[],@(x)sum(diff(x)))

B =
```

```
     0    -1     0     0
     0     0     0     0
     0     0     0     0
     2     0     0     0
```

The order of the subscripts matters:

```
val = 101:106;
subs=[1 2; 3 1; 1 2; 4 4; 4 1; 4 1];
B1 = accumarray(subs,val,[],@(x)sum(diff(x)))

B1 =

     0    -2     0     0
     0     0     0     0
     0     0     0     0
    -1     0     0     0
```

Example 3

Create a 2-by-3-by-2 array and sum values for repeated 3-D subscripts:

```
val = 101:105;
subs = [1 1 1; 2 1 2; 2 3 2; 2 1 2; 2 3 2];

A = accumarray(subs, val)
A(:,:,1) =
   101     0     0
     0     0     0
A(:,:,2) =
     0     0     0
   206     0   208
```

Example 4

Create a 2-by-3-by-2 array, and sum values natively:

```
val = 101:105;
subs = [1 1 1; 2 1 2; 2 3 2; 2 1 2; 2 3 2];
```

```
A = accumarray(subs, int8(val), [], @(x) sum(x,'native'))
A(:,:,1) =
  101    0    0
    0    0    0
A(:,:,2) =
    0    0    0
  127    0  127

class(A)
ans =
    int8
```

Example 5

Pass multiple subscript arguments in a cell array.

1. Create a 12-element vector `V`:

   ```
   V = 101:112;
   ```

2. Create three 12-element vectors, one for each dimension of the resulting array `A`. Note how the indices of these vectors determine which elements of `V` are accumulated in `A`:

   ```
   %        index 1   index 6 => V(1)+V(6) => A(1,3,1)
   %          |         |
   rowsubs = [1 3 3 2 3 1 2 2 3 3 1 2];
   colsubs = [3 4 2 1 4 3 4 2 2 4 3 4];
   pagsubs = [1 1 2 2 1 1 2 1 1 1 2 2];
   %               |
   %              index 4 => V(4) => A(2,1,2)
   %
   % A(1,3,1) = V(1) + V(6) = 101 + 106 = 207
   % A(2,1,2) = V(4) = 104
   ```

3. Call `accumarray`, passing the subscript vectors in a cell array:

   ```
   A = accumarray({rowsubs colsubs pagsubs}, V)
   ```

```
A(:,:,1) =
     0     0   207     0        % A(1,3,1) is 207
     0   108     0     0
     0   109     0   317
A(:,:,2) =
     0     0   111     0
   104     0     0   219        % A(2,1,2) is 104
     0   103     0     0
```

Example 6

Create an array with the `max` function, and fill all empty elements of that array with `NaN`:

```
val = 101:105;
subs = [1 1; 2 1; 2 3; 2 1; 2 3];

A = accumarray(subs, val, [2 4], @max, NaN)
A =
   101   NaN   NaN   NaN
   104   NaN   105   NaN
```

Example 7

Create a sparse matrix using the `prod` function:

```
val = 101:105;
subs = [1 1; 2 1; 2 3; 2 1; 2 3];

A = accumarray(subs, val, [2 4], @prod, 0, true)
A =
   (1,1)           101
   (2,1)         10608
   (2,3)         10815
```

Example 8

Count the number of entries accumulated in each bin:

```
val = 1;
```

```
subs = [1 1; 2 1; 2 3; 2 1; 2 3];

A = accumarray(subs, val, [2 4])
A =
     1     0     0     0
     2     0     2     0
```

Example 9

Create a logical array that shows which bins will accumulate two or more values:

```
val = 101:105;
subs = [1 1; 2 1; 2 3; 2 1; 2 3];

A = accumarray(subs, val, [2 4], @(x) length(x) > 1)
A =
     0     0     0     0
     1     0     1     0
```

Example 10

Group values in a cell array:

```
val = 101:105;
subs = [1 1; 2 1; 2 3; 2 1; 2 3];

A = accumarray(subs, val, [2 4], @(x) {x})
A =
    [       101]     []                []     []
    [2x1 double]     []    [2x1 double]     []

A{2}
ans =
   104
   102
```

See Also

`full` | `sparse` | `sum`

Purpose

Inverse cosine in radians

Syntax

```
Y = acos(X)
```

Description

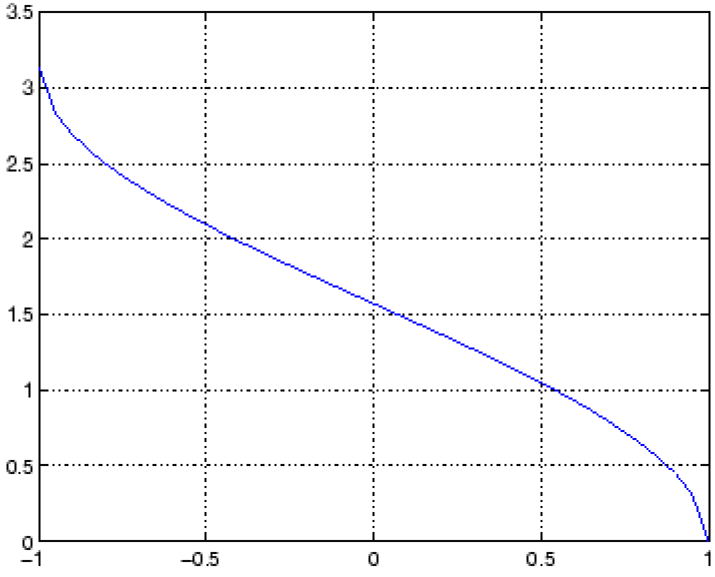
`Y = acos(X)` returns the inverse cosine (arccosine) for each element of `X`. For real elements of `X` in the domain [–1, 1], `acos(X)` is real and in the range [0, π]. For real elements of `X` outside the domain [–1, 1], `acos(X)` is complex.

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

Examples

Graph the inverse cosine function over the domain $-1 \leq x \leq 1$.

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



See Also

`acosd` | `acosh` | `cos`

Purpose

Inverse cosine in degrees

Syntax

`Y = acosd(X)`

Description

`Y = acosd(X)` returns the inverse cosine of the elements of `X` in degrees.

Input Arguments

X - Cosine of angle

scalar value | vector | matrix | N-D array

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

Data Types

`single | double`

Complex Number Support: Yes

Output Arguments

Y - Angle in degrees

scalar value | vector | matrix | N-D array

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

Examples

Inverse Cosine of Zero

Verify that inverse cosine of 0 is exactly 90.

```
acosd(0)
```

```
ans =

    90
```

Round Trip Calculation for Complex Angles

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

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

```
ans =

    2.0000    3.0000
```

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

See Also

`cosd` | `acos` | `cos`

Purpose

Inverse hyperbolic cosine

Syntax

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

Description

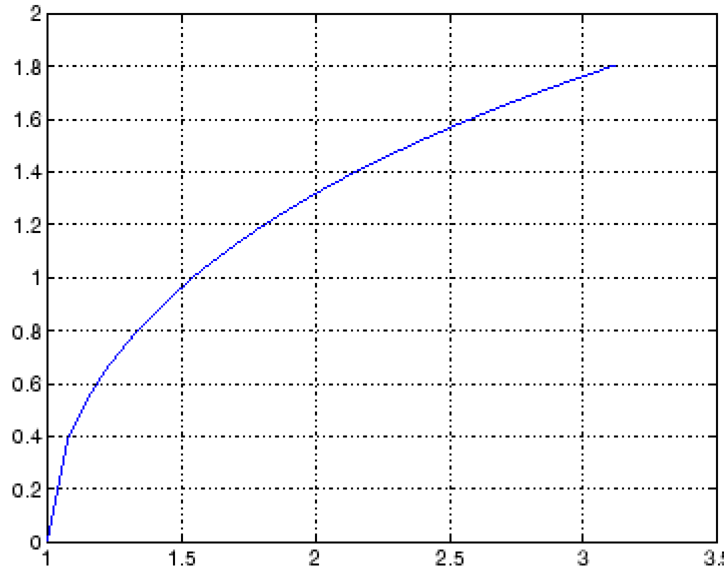
`Y = acosh(X)` returns the inverse hyperbolic cosine for each element of `X`.

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

Examples

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

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



See Also

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

acot

Purpose

Inverse cotangent in radians

Syntax

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

Description

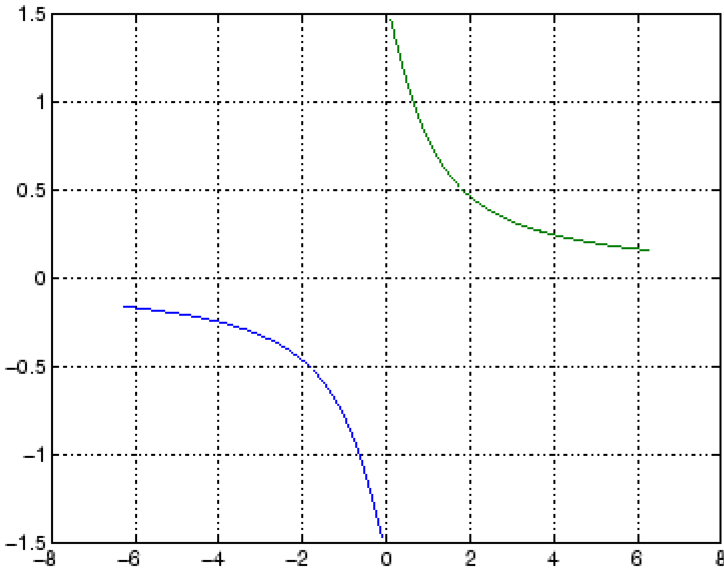
`Y = acot(X)` returns the inverse cotangent (arccotangent) for each element of `X`.

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

Examples

Graph the inverse cotangent over the domains $-2\pi \leq x < 0$ and $0 < x \leq 2\pi$.

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



See Also

`cot` | `acotd` | `acoth`

Purpose

Inverse cotangent in degrees

Syntax

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

Description

`Y = acotd(X)` returns the inverse cotangent of the elements of `X` in degrees.

Input Arguments

X - Cotangent of angle

scalar value | vector | matrix | N-D array

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

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

Output Arguments

Y - Angle in degrees

scalar value | vector | matrix | N-D array

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

Examples

Inverse Cotangent of Vector

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

```
y =

   90.0000    2.8624         0
```

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

Inverse Cotangent of Complex Value

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

```
ans =

  31.7175 -23.0535i
```

See Also

cotd | cot | acot

Purpose

Inverse hyperbolic cotangent

Syntax

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

Description

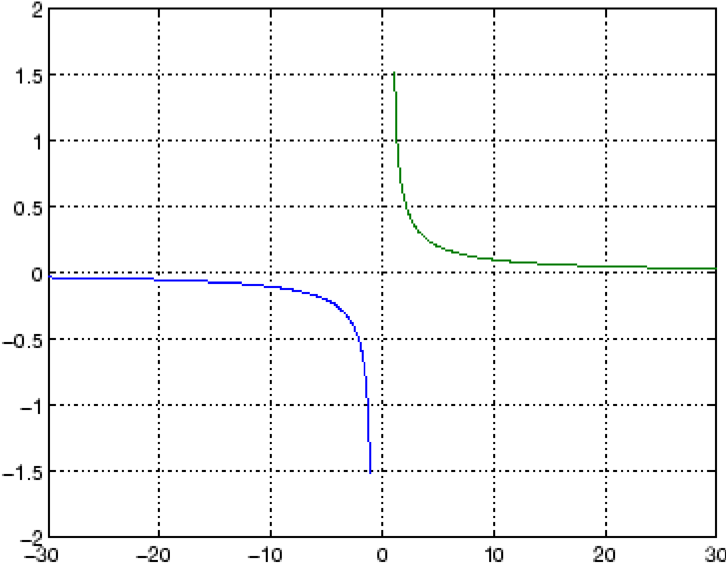
`Y = acoth(X)` returns the inverse hyperbolic cotangent for each element of `X`.

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

Examples

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

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



See Also

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

Purpose

Inverse cosecant in radians

Syntax

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

Description

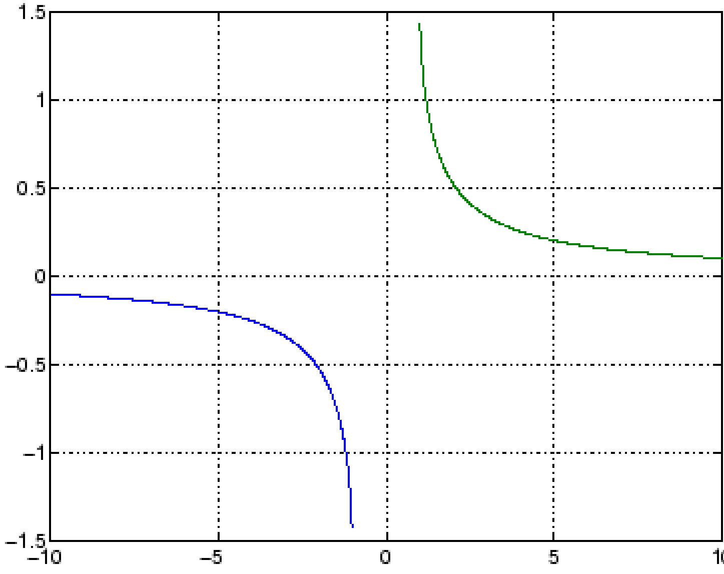
`Y = acsc(X)` returns the inverse cosecant (arccosecant) for each element of `X`.

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

Examples

Graph the inverse cosecant over the domains $-10 \le x < -1$ and $1 < x \le 10$.

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



See Also

`csc` | `acscd` | `acsch`

Purpose

Inverse cosecant in degrees

Syntax

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

Description

`Y = acscd(X)` returns the inverse cosecant of the elements of `X` in degrees.

Input Arguments

X - Cosecant of angle

scalar value | vector | matrix | N-D array

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

Data Types

`single | double`

Complex Number Support: Yes

Output Arguments

Y - Angle in degrees

scalar value | vector | matrix | N-D array

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

Examples

Inverse Cosecant of Vector

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

y =

    2.8660    5.7392         0
```

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

Inverse Cosecant of Complex Value

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

```
ans =

  25.9136 -30.4033i
```

See Also

cscd | csc | acsc

Purpose

Inverse hyperbolic cosecant

Syntax

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

Description

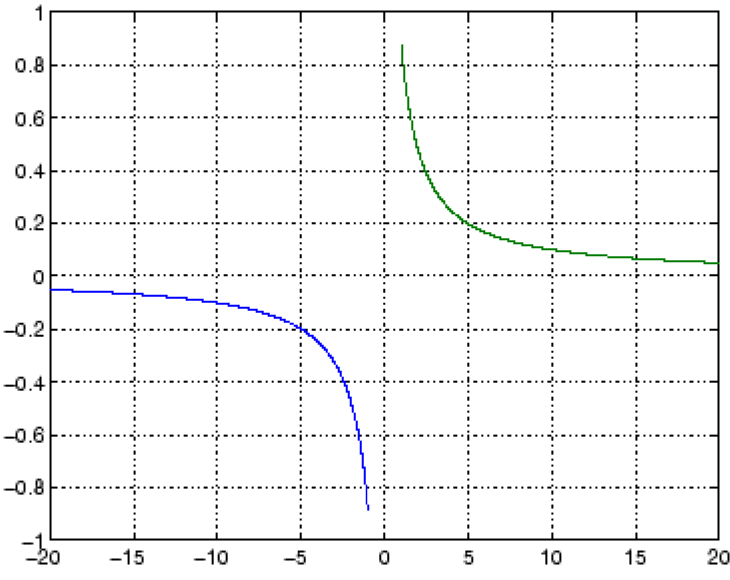
`Y = acsch(X)` returns the inverse hyperbolic cosecant for each element of `X`.

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

Examples

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

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



See Also

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

actxcontrol

Purpose

Create Microsoft ActiveX control in figure window

Syntax

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

Description

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

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

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

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

One possible format is:

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

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

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

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

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

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

`h = actxcontrol('progid', position, fig_handle)` creates an ActiveX control at the specified `position` in an existing figure window. This window is identified by the Handle Graphics® handle, `fig_handle`.

The current figure handle is returned by the `gcf` command.

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

`h = actxcontrol('progid',position,fig_handle,event_handler)` creates an ActiveX control that responds to events. Controls respond to events by invoking a MATLAB function whenever an event (such

as clicking a mouse button) is fired. The `event_handler` argument identifies one or more functions to be used in handling events. For more information, see "Specifying Event Handlers" on page 1-110.

`h = actxcontrol('progid',position,fig_handle,event_handler,'filename')` creates an ActiveX control with the first four arguments, and sets its initial state to that of a previously saved control. MATLAB loads the initial state from the file specified in the string `filename`.

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

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

Specifying Event Handlers

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

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

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

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

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

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

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

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

Event handler functions should accept a variable number of arguments.

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

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

Tips

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

When you no longer need the control, call `release` to release the interface and free memory and other resources used by the interface. Note that releasing the interface does not delete the control itself. Use the `delete` function to do this.

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

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

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

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

Examples

Handling Events

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

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

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

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

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

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

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

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

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

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

See Also

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

actxcontrollist

Purpose

List currently installed Microsoft ActiveX controls

Syntax

```
info = actxcontrollist
```

Description

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

COM functions are available on Microsoft Windows systems only.

Examples

Show information for two controls:

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

MATLAB displays information like:

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

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

See Also

`actxcontrolselect | actxcontrol`

Purpose

Create Microsoft ActiveX control from GUI

Syntax

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

Description

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

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

COM functions are available on Microsoft Windows systems only.

See Also

`actxcontrollist` | `actxcontrol`

How To

- “Creating Control Objects Using a GUI”

actxGetRunningServer

Purpose

Handle to running instance of Automation server

Syntax

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

Description

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

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

COM functions are available on Microsoft Windows systems only.

Examples

Get a handle to the MATLAB application:

```
h = actxGetRunningServer('matlab.application')
```

See Also

`actxcontrol` | `actxserver`

How To

- "MATLAB COM Automation Server Interface"

Purpose

Create COM server

Syntax

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

Description

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

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

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

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

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

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

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

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

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

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

Tips

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

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

You can register events for COM servers.

COM functions are available on Microsoft Windows systems only.

Examples

Microsoft Excel® Workbook Example

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

Create a COM server running Microsoft Excel.

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

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

Make the Excel frame window visible

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

Use the get method on the Excel object "e" to list all properties of the application:

```
e.get
```

Properties

```
        Application: [1x1
Interface.Microsoft_Excel_9.0_Object_Library._Application]
            Creator: 'xlCreatorCode'
                   .
                   .
                   .
          Workbooks: [1x1
Interface.Microsoft_Excel_9.0_Object_Library.Workbooks]
                   .
                   .
                   .
            Caption: 'Microsoft Excel - Book1'
    CellDragAndDrop: 0
   ClipboardFormats: {3x1 cell}
                   .
                   .
                   .
             Cursor: 'xlNorthwestArrow'
                   .
                   .
                   .
```

Create an interface "eWorkBooks":

```
eWorkbooks = e.Workbooks

eWorkbooks =
```

```
   Interface.Microsoft_Excel_9.0_Object_Library.Workbooks
```

List all methods for that interface

```
eWorkbooks.invoke
```

Methods

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

Add a new workbook "w", also creating a new interface:

```
w = eWorkbooks.Add

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

Close Excel and delete the object

```
e.Quit;
e.delete;
```

See Also

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

Purpose

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

Syntax

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

Description

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

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

Input Arguments

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

string

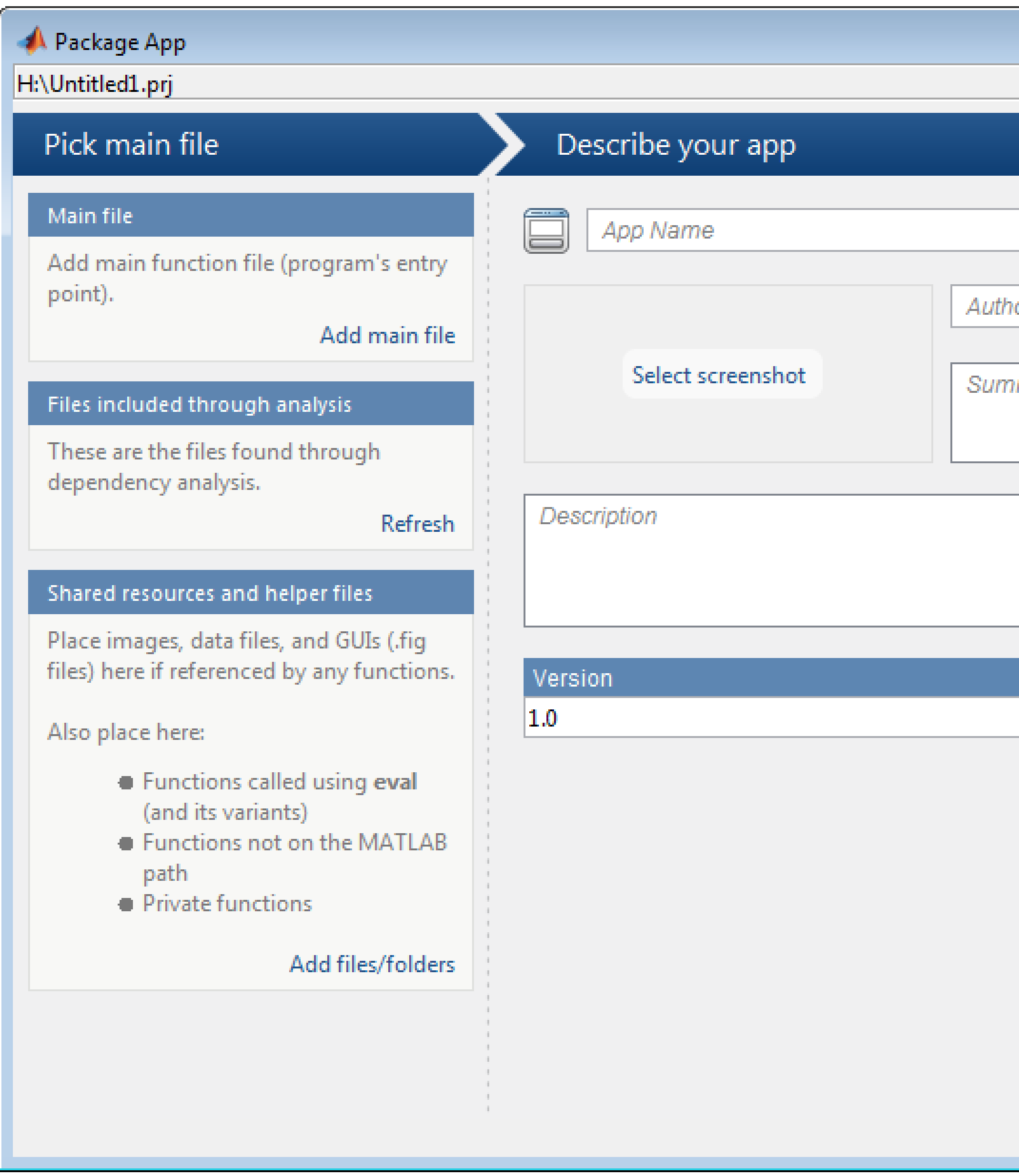
Full or partial path to the `.prj` file you created previously with the Package App dialog box, specified as a string

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

Examples

Open Dialog Box for Creating an App Package

```
matlab.apputil.create
```


Package App
H:\Untitled1.prj
Pick main file
Describe your app
Main file
Add main function file (program's entry point).
Add main file
Files included through analysis
These are the files found through dependency analysis.
Refresh
Shared resources and helper files
Place images, data files, and GUIs (.fig files) here if referenced by any functions.
Also place here:
Functions called using eval (and its variants)
Functions not on the MATLAB path
Private functions
Add files/folders
App Name
Select screenshot
Description
Version
1.0

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

Update Existing App Package

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

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

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

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

See Also

`matlab.apputil.package`

Concepts

- “MATLAB App Installer File — mlappinstall”

matlab.apputil.getInstalledAppInfo

Purpose

List installed app information

Syntax

```
matlab.apputil.getInstalledAppInfo

appinfo = matlab.apputil.getInstalledAppInfo
```

Description

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

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

Output Arguments

appinfo - Information about installed apps
structure array

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

status - Installation status
`'installed'`

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

id - Unique identifier for the installed app
string

Unique identifier for the installed app, returned as a string

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

location - Folder where the app is installed
string

Folder where the app is installed, returned as a string

name - Name of the installed app
string

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

Examples

Display Installed Apps Information in the Command Window

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

```
matlab.apputil.getInstalledAppInfo
```

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

Store Installed App Information in a Variable

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

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

Store Installed App Information in a Variable and Display IDs

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

```
myappinfo = matlab.apputil.getInstalledAppInfo

myappinfo =

1x2 struct array with fields:
    id
```

```
    name
    status
    location
```

Get the id of each installed app:

```
appids={myappinfo.id}

appids =

    'LinePlotterAPP'    'PlotRandNumbersAPP'
```

See Also

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

Concepts

- "MATLAB App Installer File — mlappinstall"

Purpose

Install app from a `.mlappinstall` file

Syntax

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

Description

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

Input Arguments

appfile - Full or partial path to `.mlappinstall` file
string

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

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

Output Arguments

appinfo - Information about installed app
structure

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

status - Installation status
`'installed'` | `'updated'`

Installation status, returned as a string:

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

id - Unique identifier
string

Unique identifier for the installed app, returned as a string

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

location - Folder where app is installed
string

Folder where app is installed, returned as a string

name - Name of installed app

string

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

Examples

Install App and Display Information About the Installation

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

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

```
appinfo = 

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

See Also

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

Concepts

- “MATLAB App Installer File — mlappinstall”

matlab.apputil.package

Purpose

Package app files into `.mlappinstall` file

Syntax

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

Description

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

Input Arguments

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

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

Example: `'plotdata.prj'`

Examples

Create mlappinstall File for Previously Created Project File

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

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

Tips

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

See Also

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

matlab.apputil.run

Purpose

Run app programmatically

Syntax

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

Description

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

Input Arguments

appid - ID of custom app

string

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

Example: `'DataExplorationAPP'`

Tips

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

Examples

Run Previously Installed App

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

Get IDs of all installed apps.

```
matlab.apputil.getInstalledAppInfo
```

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

Run `PlotData`.

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

See Also

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

matlab.apputil.uninstall

Purpose

Uninstall app

Syntax

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

Description

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

Input Arguments

appid - ID of app
string

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

Example: `'DataExplorationAPP'`

Tips

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

Examples

Uninstall App

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

View the IDs of all apps

```
matlab.apputil.getInstalledAppInfo
```

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

Uninstall the `simplegui` app.

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

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

```
matlab.apputil.getInstalledAppInfo
```

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

See Also

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

addCause (MException)

Purpose

Record additional causes of exception

Syntax

`baseExcep = addCause(baseExcep, causeExcep)`

Description

`baseExcep = addCause(baseExcep, causeExcep)` adds information to existing exception *baseExcep* to help determine its cause. The added information is in the form of a second exception *causeExcep*. Both *baseExcep* and *causeExcep* are objects of the `MException` class. The `baseExcep` and `causeExcep` inputs are scalar objects of the `mException` class.

The exception has a property called `cause` in which you can store a series of additional exceptions, each saving information on what caused the initial error. (See the figure in the documentation for "The MException Class".) When your program calls `addCause`, MATLAB appends a new exception *cause* to this field in the base exception *exception*. When your error handling code catches the error in a `try/catch` statement, execution of the `catch` part of this statement makes the base exception, along with all of the appended cause records, available to help diagnose the error.

Examples

This example attempts to open a file in a folder that is not on the MATLAB path. It uses a nested `try-catch` block to give the user the opportunity to extend the path. If the file still cannot be found, the program issues an exception with the first error appended to the second using `addCause`:

```
function data = read_it(filename);
try
   % Attempt to open and read from a file.
   fid = fopen(filename, 'r');
   data = fread(fid);
catch exception1
   % If the error was caused by an invalid file ID, try
   % reading from another location.
   if strcmp(exception1.identifier, 'MATLAB:FileIO:InvalidFid')
      msg = sprintf( ...
```

```
            '\nCannot open file %s. Try another location?  ', ...
            filename);
         reply = input(msg, 's')
         if reply(1) == 'y'
             newFolder = input('Enter folder name:  ', 's');
         else
             throw(exception1);
         end
         oldpath = addpath(newFolder);
         try
            fid = fopen(filename, 'r');
            data = fread(fid);
         catch exception2
            exception3 = addCause(exception2, exception1)
            path(oldpath);
            throw(exception3);
         end
         path(oldpath);
      end
   end
   fclose(fid);

   try
      d = read_it('anytextfile.txt');
   catch exception
   end

   exception
   exception =
    MException object with properties:

       identifier: 'MATLAB:FileIO:InvalidFid'
          message: 'Invalid file identifier.  Use fopen
                    to generate a valid file identifier.'
            stack: [1x1 struct]
            cause: {[1x1 MException]}
```

```
  Cannot open file anytextfile.txt. Try another location?y
Enter folder name:  xxxxxxx
Warning: Name is nonexistent or not a directory: xxxxxxx.
> In path at 110
  In addpath at 89
```

See Also

try | catch | error | assert | MException | throw(MException) | rethrow(MException) | throwAsCaller(MException) | getReport(MException) | last(MException)

Purpose

Add event to `timeseries` object

Syntax

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

Description

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

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

Examples

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

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

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

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

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

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

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

```
get(e1)
```

MATLAB software responds with

```
    EventData: []
```

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

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

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

See Also

`timeseries` | `tsdata.event`

Purpose

Information about audio file

Syntax

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

Description

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

Input Arguments

filename - Name of file
string

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

Example: `'myFile.mp3'`

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

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

`audioinfo` supports the following file formats.

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

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

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

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

Output Arguments

info - Information about audio file

structure

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

Field Name	Description	Data Type
`Filename`	Filename including the absolute path to the file and the file extension.	`string`
`CompressionMethod`	Compression method used.	`string`
`NumChannels`	Number of audio channels encoded in	`double`
`SampleRate`	Sample rate of the audio data in the file, in hertz.	`double`
`TotalSamples`	Total number of audio samples in the file.	`double`
`Duration`	Duration of the file, in seconds.	`double`
`BitsPerSample`	Number of bits per sample encoded in the audio file. Only valid for WAVE (`.wav`) and FLAC (`.flac`) files.	`double`

Field Name	Description	Data Type
`BitRate`	Number of kilobits per second (kbit/s) used for compressed audio files. Only valid for MP3 (`.mp3`) and MPEG-4 Audio (`.m4a`, `.mp4`) files.	`double`
`Title`	Value of `'Title'`, if any.	`string`
`Artist`	Value of `'Artist'`, if any.	`string`
`Comment`	Value of `'Comment'`, if any.	`string`

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

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

Examples

Get Information About Audio File

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

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

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

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

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

```
info =
             Filename: 'S:\handel.wav'
    CompressionMethod: 'Uncompressed'
          NumChannels: 1
           SampleRate: 8192
         TotalSamples: 73113
             Duration: 8.9249
                Title: []
              Comment: []
               Artist: []
        BitsPerSample: 16
```

Limitations

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

See Also

`audioread` | `audiowrite`

Purpose

Read audio file

Syntax

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

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

Description

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

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

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

Input Arguments

filename - Name of file to read

string

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

Example: `'myFile.mp3'`

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

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

`audioread` supports the following file formats.

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

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

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

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

samples - Audio samples to read

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

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

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

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

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

Example: `[1,100]`

Data Types
`double`

dataType - Data format of audio data, `y`

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

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

`'double'`	Double-precision normalized samples.
`'native'`	Samples in the native format found in the file.

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

Output Arguments

y - Audio data

matrix

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

- If you do not specify `dataType`, or `dataType` is `'double'`, then `y` is of type `double`, and matrix elements are normalized values between −1.0 and 1.0.

- If `dataType` is `'native'`, then `y` can be one of several MATLAB data types, depending on the file format and the `BitsPerSample` value of the input file. Call `audioinfo` to determine the `BitsPerSample` value of the file.

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

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

Fs - Sample rate
positive scalar

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

Examples

Read Complete Audio File

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

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

```
load handel.mat

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

Read the data back into MATLAB using `audioread`.

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

Play the audio.

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

Read Portion of Audio File

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

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

```
load handel.mat
```

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

Read only the first 2 seconds.

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

Play the samples.

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

Return Audio in Native Integer Format

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

```
whos y
```

```
  Name           Size                Bytes  Class     Attributes

  y          16384x1                131072  double
```

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

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

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

```
  Name           Size               Bytes  Class    Attributes
```

```
  y          16384x1              32768  int16
```

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

Limitations

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

See Also

`audiowrite` | `audioinfo`

audiowrite

Purpose

Write audio file

Syntax

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

Description

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

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

Input Arguments

filename - Name of file to write

string

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

`audiowrite` supports the following file formats.

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

Example: `'myFile.m4a'`

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

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

When writing AAC files on Windows, `audiowrite` pads the front and back of the output signal with extra samples of silence. The Windows AAC encoder also places a very sharp fade-in and fade-out on the audio.

This results in audio with an increased number of samples after being written to disk.

y - Audio data to write

matrix

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

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

The maximum number of channels depends on the file format.

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

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

Data Type of y	Valid Range for y
uint8	0 ≤ y ≤ 255
int16	-32768 ≤ y ≤ +32767
int32	-2^32 ≤ y ≤ 2^32–1
single	-1.0 ≤ y ≤ +1.0
double	-1.0 ≤ y ≤ +1.0

Data beyond the valid range is clipped.

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

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

`Fs` - Sample rate

positive scalar

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

Example: `44100`

Data Types
`double`

Name-Value Pair Arguments

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

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

'BitsPerSample' - Number of output bits per sample

16 (default) | 8 | 24 | 32 | 64

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

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

Example: `'BitsPerSample',32`

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

'BitRate' - Kilobits per second (kbit/s)

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

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

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

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

Example: `'BitRate',96`

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

'Quality' - Quality setting for the Ogg Vorbis Compressor

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

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

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

Example: `'Quality',25`

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

'Title' - Title information

`[]` (default) | string

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

'Artist' - Artist information

`[]` (default) | string

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

'Comment' - Additional information
`[]` (default) | string

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

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

Examples

Write an Audio File

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

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

```
load handel.mat

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

Read the data back into MATLAB using `audioread`.

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

Listen to the audio.

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

Specify Bits Per Sample and Metadata

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

```
load handel.mat

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

View information about the new FLAC file.

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

```
info =

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

Algorithms

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

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

See Also

audioinfo | audioread

Purpose

Add frame to Audio/Video Interleaved (AVI) file

Syntax

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

Description

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

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

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

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

`aviobj = addframe(aviobj,h)` captures a frame from the figure or axis handle `h` and appends this frame to the AVI file. `addframe` renders the figure into an offscreen array before appending it to the AVI file. This ensures that the figure is written correctly to the AVI file even if the figure is obscured on the screen by another window or screen saver.

Tips

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

the syntax *aviobj* = addframe(*aviobj*,*mov*). See the example for an illustration.

Examples

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

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

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

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

See Also

VideoWriter | avifile | close (avifile) | movie2avi

Purpose

Create event listener

Syntax

```
lh = addlistener(Hsource,'EventName',callback)
lh = addlistener(Hsource,property,'EventName',callback)
```

Description

`lh = addlistener(Hsource,'EventName',callback)` creates a listener for the specified event.

`lh = addlistener(Hsource,property,'EventName',callback)` creates a listener for one of the predefined property events. There are four property events:

- `PreSet` — triggered just before the property value is set, before calling its set access method.
- `PostSet` — triggered just after the property value is set.
- `PreGet` — triggered just before a property value query is serviced, before calling its get access method.
- `PostGet` — triggered just after returning the property value to the query

See "Events and Listeners — Syntax and Techniques" for more information.

Input Arguments

`Hsource`
: Handle of the object that is the source of the event, or an array of source handles.

`EventName`
: Name of the event, which is triggered by the source objects.

`callback`
: Function handle referencing a function to execute when the event is triggered.

`property`
: Character string that can be:

  - the name of the property

- a cell array of strings where each string is the name of a property that exists in object array `Hsource`
- a `meta.property` object or an array of `meta.property` objects
- a cell array of `meta.property` objects

If `Hsource` is a scalar, then any of the properties can be dynamic properties. If `Hsource` is non-scalar, then the properties must belong to the class of `Hsource` and can not include dynamic properties (which are not part of the class definition).

For more information, see the following sections:

- The `GetObservable` and `SetObservable` property attributes in the "Property Attributes" table.
- "Creating Property Listeners"
- "Dynamic Properties — Adding Properties to an Instance"

Output Arguments

`lh`

Handle of the `event.listener` object returned by `addlistener`.

Removing a Listener

To remove a listener, delete the listener object returned by `addlistener`. For example,

```
delete(lh)
```

calls the handle class delete method to delete the object from the workspace and remove the listener.

When you use `addlistener` to create a listener, redefining the variable containing the handle to the listener does not delete the listener because the event object still has a reference to the `event.listener` object. `addlistener` ties the listener's lifecycle to the object that is the source of the event. If you want to define a listener that is not tied to the event object, use the `event.listener` constructor directly to create the listener.

See Also

`delete (handle)` | `handle` | `notify (handle)`

How To

- "Limiting Listener Scope — Constructing event.listener Objects Directly"

inputParser.addOptional

Purpose

Add optional argument to Input Parser scheme

Syntax

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

Description

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

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

Tips

- For optional string inputs, specify a validation function. Without a validation function, the parser interprets valid string inputs as invalid parameter names and throws an error.

Input Arguments

p

Object of class `inputParser`.

argName

String that specifies the internal name for the input argument.

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

- You must specify inputs in the order that they are added to the input parser scheme.
- If you want to specify a value for the `k`th argument, you must also specify values for the first `(k 1)` arguments in the input scheme.

default

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

validationFcn

Handle to a function that checks whether the input argument is of the expected type and value.

Validation functions must accept a single input argument, and they must either return a scalar logical value (`true` or `false`) or an error. If the validation function returns `false` or errors, then parsing fails and the parser throws an error.

Examples

Add Optional Input

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

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

Validate Optional Input

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

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

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

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

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

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

See Also

`addParamValue` | `addRequired` | `inputParser` | `function_handle`

Concepts

- “Anonymous Functions”

Purpose

Add parameter name and value argument to Input Parser scheme

Syntax

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

Description

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

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

Tips

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

Input Arguments

p

Object of class `inputParser`.

paramName

String that specifies the internal name for the input parameter.

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

default

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

validationFcn

Handle to a function that checks whether the input argument is of the expected type and value.

Validation functions must accept a single input argument, and they must either return a scalar logical value (`true` or `false`) or an error. If the validation function returns `false` or errors, then parsing fails and the parser throws an error.

Examples

Add Optional Parameter Value Input

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

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

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

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

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

```
ans =
    myparam: 100
```

Validate Parameter Value

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

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

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

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

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

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

See Also

addOptional | addRequired | inputParser | function_handle

Concepts

- “Anonymous Functions”

addpath

Purpose

Add folders to search path

Syntax

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

addpath( ___ ,'-frozen')

oldpath = addpath( ___ )
```

Description

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

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

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

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

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

Input Arguments

folderName1,...,folderNameN - Names of folders to add to search path

string

Names of folders to add to the search path, specified as strings. Use the full path name for each folder. Use `genpath` with `addpath` to add all subfolders of `folderName`.

Example: `'c:\matlab\work'`

Example: `'/home/user/matlab'`

Example: `'/home/user/matlab','/home/user/matlab/test'`

position - Position on the search path
`'-begin'` (default) | `'-end'`

Position on the search path, specified as one of the following strings.

Value of `position`	Description
`'-begin'`	Add specified folders to the top of the search path.
`'-end'`	Add specified folders to the bottom of the search path.

Output Arguments

oldpath - Path prior to addition of folders
string

Path prior to the addition of folders, returned as a string.

Examples

Add Folder to Top of Search Path

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

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

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

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

Add Folder to End of Search Path

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

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

Add Folder and Its Subfolders to Search Path

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

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

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

Add Folder to Search Path and Disable Folder Change Notification

On Windows, add the folder `c:/matlab/myfiles` to the top of the search path, disable folder change notification, and return the search path before adding the folder.

```
oldpath = addpath('c:/matlab/myfiles','-frozen');
```

Tips

- If you use `addpath` within a local function, the path change persists after program control returns from the function. That is, the scope of the path change is global.

See Also

`genpath` | `path` | `pathsep` | `rmpath` | `savepath`

Concepts

- "What Is the MATLAB Search Path?"
- "Files and Folders that MATLAB Accesses"
- "Path Names in MATLAB"
- "Specifying Startup Options in the MATLAB Startup File"

Purpose

Add preference

Syntax

```
addpref('group','pref',val)
addpref('group',{'pref1','pref2',...'prefn'},{val1,val2,
   ...valn})
```

Description

`addpref('group','pref',val)` creates the preference specified by `group` and `pref` and sets its value to `val`. It is an error to add a preference that already exists. Individual preference values can be any MATLAB data type, including numeric types, strings, cell arrays, structures, and objects.

`group` labels a related collection of preferences. You can choose any name that is a legal variable name, and is descriptive enough to be unique, e.g. `'ApplicationOnePrefs'`. The input argument `pref` identifies an individual preference in that group, and must be a legal variable name.

`addpref('group',{'pref1','pref2',...'prefn'},{val1,val2,...valn})` creates the preferences specified by the cell array of names `'pref1'`, `'pref2',...,'prefn'`, setting each to the corresponding value.

Note Preference values are persistent and maintain their values between MATLAB sessions. Where they are stored is system dependent.

Examples

Add a preference called `version` to the `mytoolbox` group of preferences, setting its value to the cell array `{'1.0','beta'}`.

```
addpref('mytoolbox','version',{'1.0','beta'})
```

Add a preference called `documentation` to the `mytoolbox` group of preferences, setting its value to a struct you define as having four fields:

```
mydoc.docexists = 1;
mydoc.docpath = fullfile(docroot,'techdoc',...
```

```
   'matlab_env','examples');
mydoc.demoexists = 1;
mydoc.demopath = fullfile(docroot,'techdoc',...
   'matlab_env','examples','demo_examples');
addpref('mytoolbox','documentation',mydoc)
% Retrieve the preference with GETPREF
p = getpref('mytoolbox','documentation')

p =
     docexists: 1
       docpath: [1x109 char]
    demoexists: 1
      demopath: [1x123 char]
```

See Also

getpref | ispref | rmpref | setpref | uigetpref | uisetpref

Purpose

Add dynamic property

Syntax

`P = addprop(Hobj,'PropName')`

Description

`P = addprop(Hobj,'PropName')` adds a property named `PropName` to each object in array `Hobj`. The class definition is not affected by the addition of dynamic properties. Note that you can add dynamic properties only to objects derived from the `dynamicprops` class. You can set and retrieve the data in dynamic properties as you would any property.

The output argument `P` is an array the same size as `Hobj` of `meta.DynamicProperty` objects, which you can use to assign `SetMethod` and `GetMethod` functions to the property. These functions operate just like property set and get access methods.

See "Dynamic Properties — Adding Properties to an Instance" for more information and examples.

See Also

`handle` | `dynamicprops`

addproperty

Purpose

Add custom property to COM object

Syntax

```
h.addproperty('propertyname')
addproperty(h,'propertyname')
```

Description

`h.addproperty('propertyname')` adds the custom property specified in the string `propertyname` to the object or interface `h`. Use the COM `set` function to assign a value to the property.

`addproperty(h,'propertyname')` is an alternate syntax.

COM functions are available on Microsoft Windows systems only.

Examples

Add a custom property to an instance of the MATLAB sample control:

1 Create an instance of the control:

```
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f);
h.get
```

MATLAB displays its properties:

```
     Label: 'Label'
    Radius: 20
```

2 Add a custom property named `Position` and assign a value:

```
h.addproperty('Position');
h.Position = [200 120];
h.get
```

MATLAB displays (in part):

```
       Label: 'Label'
      Radius: 20
    Position: [200 120]
```

3 Delete the custom property `Position`:

```
h.deleteproperty('Position');
h.get
```

MATLAB displays the original list of properties:

```
  Label: 'Label'
 Radius: 20
```

See Also

deleteproperty | get (COM) | set (COM) | inspect

How To

- "Use Object Properties"

inputParser.addRequired

Purpose

Add required argument to Input Parser scheme

Syntax

```
addRequired(p,argName)
addRequired(p,argName,validationFcn)
```

Description

`addRequired(p,argName)` adds required argument `argName` to the input scheme of `inputParser` object `p`.

`addRequired(p,argName,validationFcn)` includes a validation function for the input argument.

Input Arguments

p

Object of class `inputParser`.

argName

String that specifies the internal name for the input argument.

Arguments added with `addRequired` are positional. When you call a function with positional inputs, you must specify inputs in the order that they are added to the input parser scheme.

validationFcn

Handle to a function that checks whether the input argument is of the expected type and value.

Validation functions must accept a single input argument, and they must either return a scalar logical value (`true` or `false`) or an error. If the validation function returns `false` or errors, then parsing fails and the parser throws an error.

Examples

Add Required Input

Create an `inputParser` object and add a required input named `myinput` to the input scheme.

```
p = inputParser;
argName = 'myinput';
addRequired(p,argName);
```

Validate Required Input

Check whether a required input named `num` is a numeric scalar greater than zero.

```
p = inputParser;
argName = 'num';
validationFcn = @(x) isnumeric(x) && isscalar(x) && (x > 0);
addRequired(p,argName,validationFcn);
```

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

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

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

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

See Also

`addOptional` | `addParamValue` | `inputParser` | `function_handle`

Concepts

- "Anonymous Functions"

addsampletocollection

Purpose

Add sample to `tscollection` object

Syntax

```
tsc = addsampletocollection(tsc,'time',Time,TS1Name,TS1Data,
   TSnName,TSnData)
```

Description

`tsc = addsampletocollection(tsc,'time',Time,TS1Name,TS1Data, TSnName,TSnData)` adds data samples `TSnData` to the collection member `TSnName` in the `tscollection` object `tsc` at one or more `Time` values. Here, `TSnName` is the string that represents the name of a time series in `tsc`, and `TSnData` is an array containing data samples.

Tips

If you do not specify data samples for a time-series member in tsc, that time-series member will contain missing data at the times given by Time (for numerical time-series data), `NaN` values, or (for logical time-series data) `false` values.

When a time-series member requires `Quality` values, you can specify data quality codes together with the data samples by using the following syntax:

```
tsc = addsampletocollection(tsc,'time',time,TS1Name,...
         ts1cellarray,TS2Name,ts2cellarray,...)
```

Specify data in the first cell array element and `Quality` in the second cell array element.

Note If a time-series member already has `Quality` values but you only provide data samples, `0`s are added to the existing `Quality` array at the times given by `Time`.

Examples

The following example shows how to create a `tscollection` that consists of two `timeseries` objects, where one `timeseries` does not have quality codes and the other does. The final step of the example adds a sample to the `tscollection`.

1 Create two `timeseries` objects, `ts1` and `ts2`.

```
ts1 = timeseries([1.1 2.9 3.7 4.0 3.0],1:5,...
                 'name','acceleration');
ts2 = timeseries([3.2 4.2 6.2 8.5 1.1],1:5,...
                 'name','speed');
```

2 Define a dictionary of quality codes and descriptions for `ts2`.

```
ts2.QualityInfo.Code = [0 1];
ts2.QualityInfo.Description = {'bad','good'};
```

3 Assign a quality of code of `1`, which is equivalent to `'good'`, to each data value in `ts2`.

```
ts2.Quality = ones(5,1);
```

4 Create a time-series collection `tsc`, which includes time series `ts1` and `ts2`.

```
tsc = tscollection({ts1,ts2});
```

5 Add a data sample to the collection `tsc` at `3.5` seconds.

```
tsc = addsampletocollection(tsc,'time',3.5,'acceleration',10,'speed',{5 1});
```

The cell array for the `timeseries` object `'speed'` specifies both the data value 5 and the quality code `1`.

Note If you do not specify a quality code when adding a data sample to a time series that has quality codes, then the lowest quality code is assigned to the new sample by default.

See Also

`delsamplefromcollection` | `timeseries` | `tscollection`

Purpose Modify date number by field

Syntax

```
R = addtodate(D, Q, F)
```

Description `R = addtodate(D, Q, F)` adds quantity `Q` to the indicated date field `F` of a scalar serial date number `D`, returning the updated date number `R`.

The quantity `Q` to be added can be a positive or negative integer. The absolute value of `Q` must be less than or equal to 1e16. The date field `F` must be a 1-by-`N` character array equal to one of the following: `'year'`, `'month'`, `'day'`, `'hour'`, `'minute'`, `'second'`, or `'millisecond'`.

If the addition to the date field causes the field to roll over, the MATLAB software adjusts the next more significant fields accordingly. Adding a negative quantity to the indicated date field rolls back the calendar on the indicated field. If the addition causes the field to roll back, MATLAB adjusts the next less significant fields accordingly.

Examples Modify the hours, days, and minutes of a given date:

```
t = datenum('07-Apr-2008 23:00:00');
datestr(t)
ans =
   07-Apr-2008 23:00:00

t= addtodate(t, 2, 'hour');
datestr(t)
ans =
   08-Apr-2008 01:00:00

t= addtodate(t, -7, 'day');
datestr(t)
ans =
   01-Apr-2008 01:00:00

t= addtodate(t, 59, 'minute');
datestr(t)
ans =
```

```
   01-Apr-2008 01:59:00
```

Adding 20 days to the given date in late December causes the calendar to roll over to January of the next year:

```
R = addtodate(datenum('12/24/2007 12:45'), 20, 'day');

datestr(R)
ans =
   13-Jan-2008 12:45:00
```

See Also

date | datenum | datestr | datevec

addts

Purpose

Add timeseries object to tscollection object

Syntax

```
tsc = addts(tsc,ts)
tsc = addts(tsc,ts)
tsc = addts(tsc,ts,Name)
tsc = addts(tsc,Data,Name)
```

Description

`tsc = addts(tsc,ts)` adds the `timeseries` object ts to `tscollection` object tsc.

`tsc = addts(tsc,ts)` adds a cell array of `timeseries` objects ts to the `tscollection` tsc.

`tsc = addts(tsc,ts,Name)` adds a cell array of `timeseries` objects ts to `tscollection` tsc. Name is a cell array of strings that gives the names of the `timeseries` objects in `ts`.

`tsc = addts(tsc,Data,Name)` creates a new `timeseries` object from Data with the name Name and adds it to the `tscollection` object tsc. Data is a numerical array and Name is a string.

Tips

The `timeseries` objects you add to the collection must have the same time vector as the collection. That is, the time vectors must have the same time values and units.

Suppose that the time vector of a `timeseries` object is associated with calendar dates. When you add this `timeseries` to a collection with a time vector without calendar dates, the time vectors are compared based on the units and the values relative to the `StartDate` property. For more information about properties, see the `timeseries` reference page.

Examples

The following example shows how to add a time series to a time-series collection:

1. Create two `timeseries` objects, `ts1` and `ts2`.

```
ts1 = timeseries([1.1 2.9 3.7 4.0 3.0],1:5,...
                 'name','acceleration');
```

```
ts2 = timeseries([3.2 4.2 6.2 8.5 1.1],1:5,...
                 'name','speed');
```

2. Create a time-series collection `tsc`, which includes `ts1`.

```
tsc = tscollection(ts1);
```

3. Add `ts2` to the `tsc` collection.

```
tsc = addts(tsc, ts2);
```

4. To view the members of `tsc`, type

```
tsc
```

at the MATLAB prompt. the response is

```
Time Series Collection Object: unnamed

Time vector characteristics

      Start time            1 seconds
      End time              5 seconds

Member Time Series Objects:

      acceleration
      speed
```

The members of `tsc` are listed by name at the bottom: `acceleration` and `speed`. These are the `Name` properties of the `timeseries` objects `ts1` and `ts2`, respectively.

See Also

`removets` | `tscollection`

Purpose

Airy Functions

Compatibility

The behavior of `airy` has changed in the following ways:

- `airy` no longer supports the syntax `[W,ierr] = airy(k,Z)`.
- `airy` no longer accepts `k` and `Z` passed as a combination of row and column vectors.
- `airy` no longer accepts `k` and `Z` passed as a combination of nonscalar and empty inputs.

For more information, see "Functionality being removed or changed".

Syntax

```
W = airy(Z)
W = airy(k,Z)
W = airy(k,Z,scale)

[W,ierr] = airy(k,Z)
```

Description

`W = airy(Z)` returns the Airy function, $Ai(Z)$, for each element of Z.

`W = airy(k,Z)` returns any of four different Airy functions, depending on the value of `k`, such as the Airy function of the second kind or the first derivative of an Airy function.

`W = airy(k,Z,scale)` scales the resulting Airy function. `airy` applies a specific scaling function to `W` depending on your choice of `k` and `scale`.

`[W,ierr] = airy(k,Z)` returns the Airy function and the completion flags in an array the same size as `W`.

Input Arguments

Z - System variable

vector | matrix | N-D Array

System variable, specified as a real or complex vector, matrix, or N-D array.

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

k - Type of Airy function

0 (default) | 1 | 2 | 3

Type of Airy function, specified as one of four values.

k	Returns
0	Airy function, $Ai(Z)$, which is the same as `airy(Z)`.
1	First derivative of Airy function, $Ai'(Z)$.
2	Airy function of the second kind, $Bi(Z)$
3	First derivative of Airy function of the second kind, $Bi'(Z)$

Data Types
`single | double`

scale - Scaling option

0 (default) | 1

Scaling option, specified as `0` or `1`. Use `scale = 1` to enable the scaling of `Z`. The values you specify for `k` and `scale` determine the scaling function `airy` applies to `Z`.

scale	k	Scaling applied to output
0	Any	None
1	0 or 1	$e^{\frac{2}{3}Z^{(3/2)}}$
1	2 or 3	$e^{-\left\|\frac{2}{3}Re\left(Z^{(3/2)}\right)\right\|}$

Data Types
`single` | `double`

Output Arguments

W - Airy function of `Z`
vector | matrix | N-D Array

Airy function of `Z`, returned as an array the same size as `Z`.

ierr - Completion flags
vector | matrix | N-D Array

Completion flags, returned as a vector, matrix, or N-D array the same size as `Z`. The elements of `ierr` are assigned values between `0` and `5`, representing successful completion or an error state.

Flag Value	Description
0	`airy` successfully computed the Airy function for this element.
1	Illegal arguments.
2	Overflow. Returns `Inf`.
3	Some loss of accuracy in argument reduction.

Flag Value	Description
4	Unacceptable loss of accuracy, `Z` too large.
5	No convergence. Returns `NaN`.

The `airy` function no longer supports `ierr`.

Definitions

Airy Functions

The Airy functions form a pair of linearly independent solutions to

$$\frac{d^2W}{dZ^2} - ZW = 0.$$

The relationship between the Airy and modified Bessel functions is

$$Ai(Z) = \left[\frac{1}{\pi}\sqrt{\frac{Z}{3}}\right]K_{1/3}(\zeta)$$

$$Bi(Z) = \sqrt{\frac{Z}{3}}\left[I_{-1/3}(\zeta) + I_{1/3}(\zeta)\right],$$

where

$$\zeta = \frac{2}{3}Z^{3/2}.$$

Examples

Airy Function of Real-Valued *x*

Define x.

```
x = -10:0.01:1;
```

Calculate $Ai(x)$.

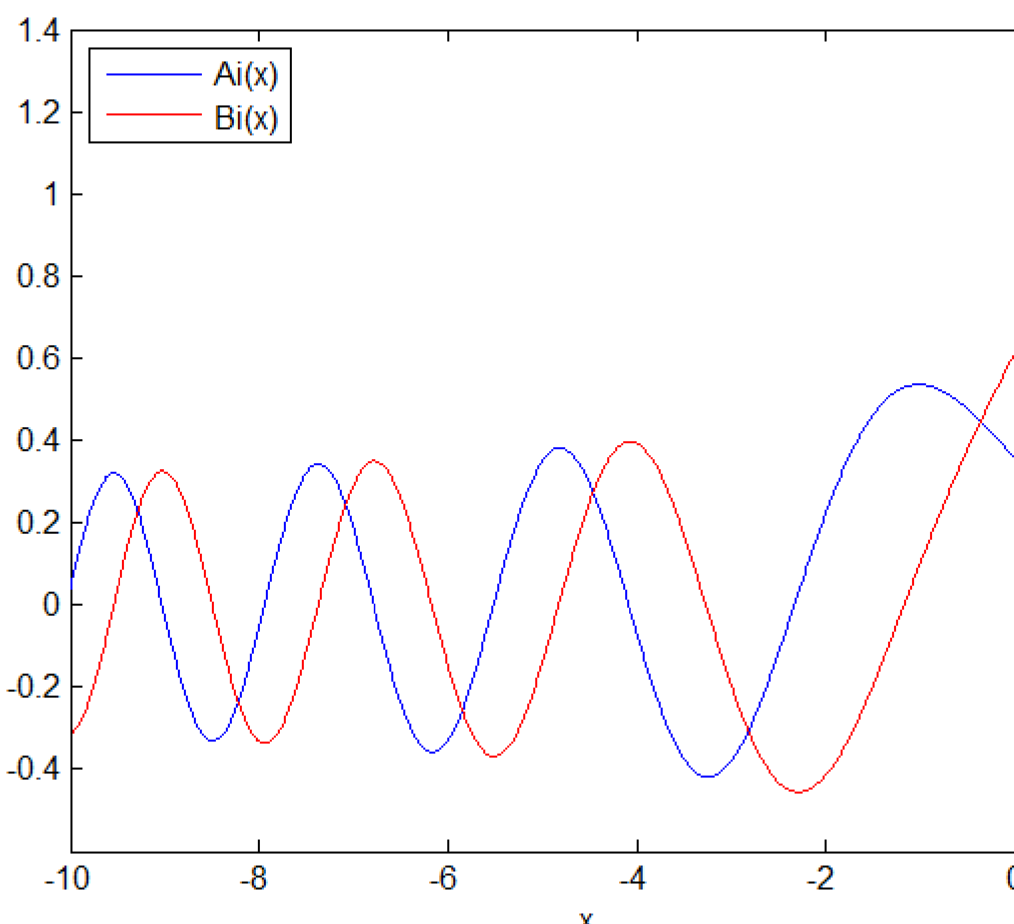
```
ai = airy(x);
```

Calculate $Bi(x)$ using `k = 2`.

```
bi = airy(2,x);
```

Plot both results together on the same axes.

```
figure
plot(x,ai,'-b',x,bi,'-r');
axis([-10 1 -0.6 1.4]);
xlabel('x');
legend('Ai(x)','Bi(x)','Location','NorthWest');
```



Airy Function of Complex-Valued Input

Compute the Airy function at a slice though the complex plane at $x+i$.

Take a slice through the complex plane.

```
x = -4:0.1:4;
```

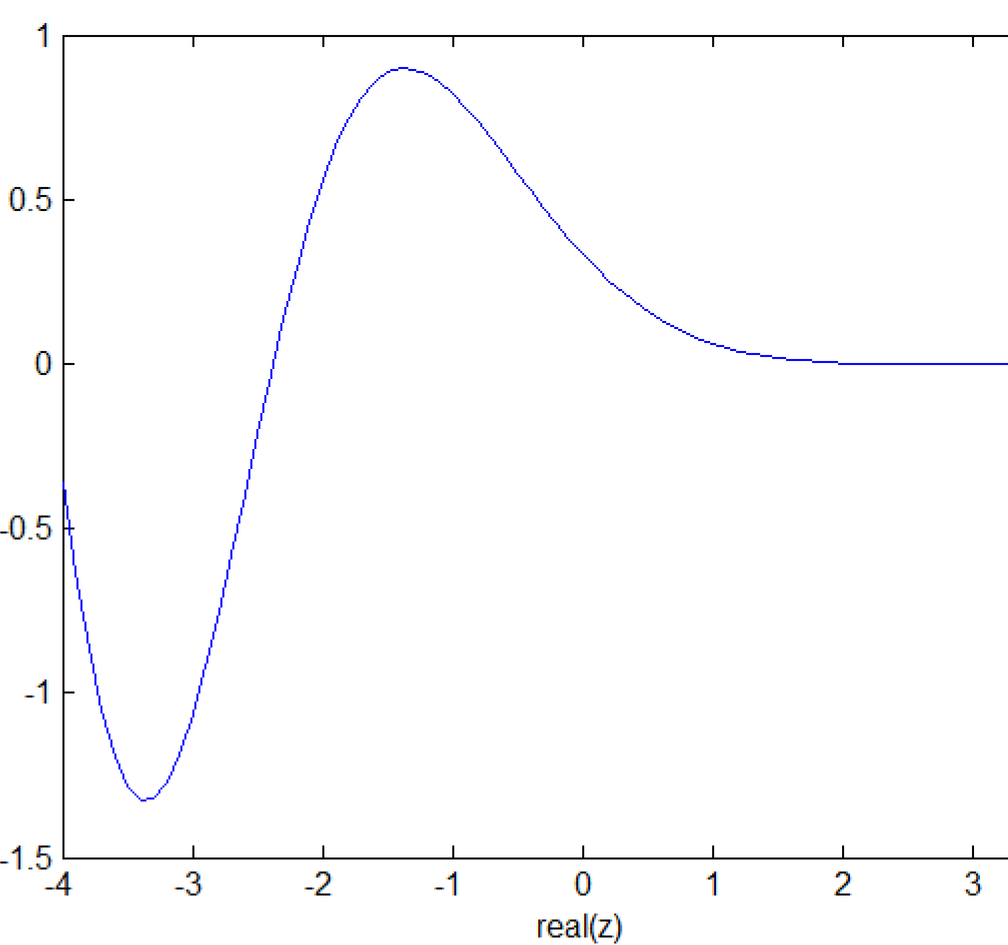
```
z = x+1i;
```

Calculate $Ai(z)$.

```
w = airy(z);
```

Plot the real part of the result.

```
figure
plot(x, real(w));
axis([-4 4 -1.5 1]);
xlabel('real(z)');
```



Scaled Airy Function

Define x.

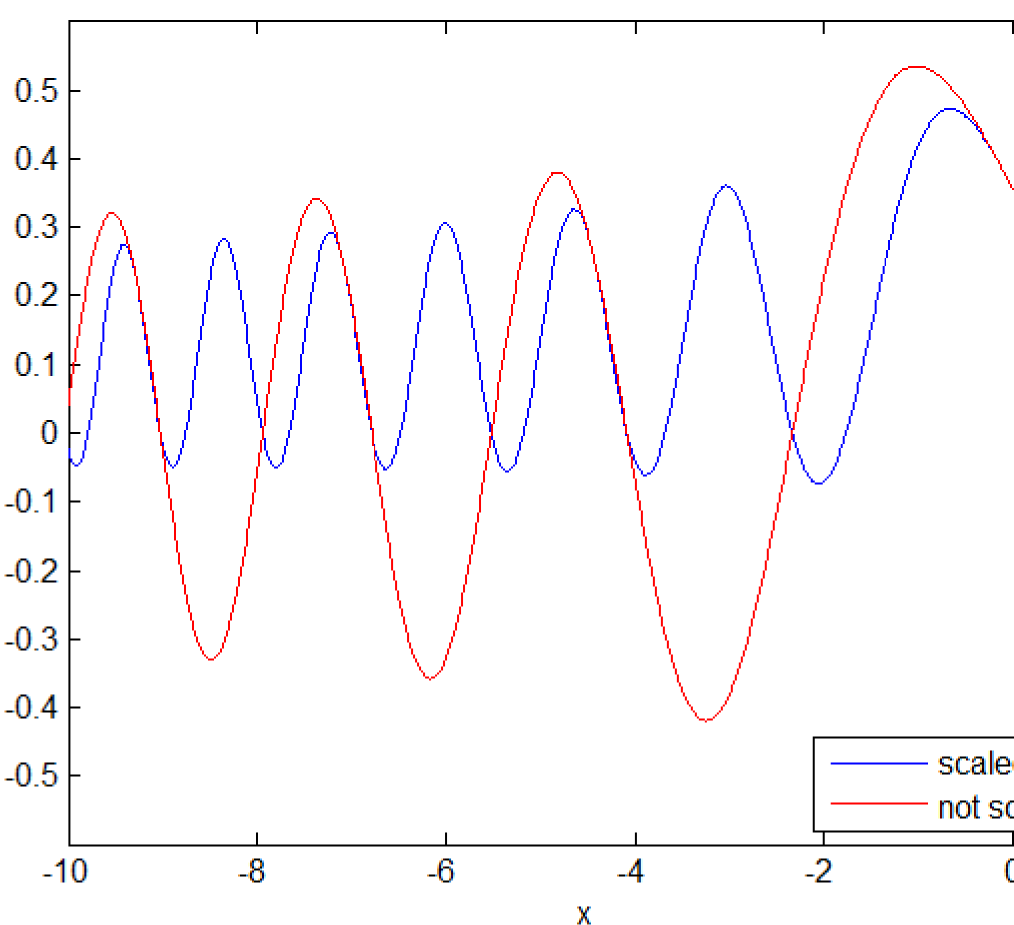
```
x = -10:0.01:1;
```

Calculate the scaled and unscaled Airy function.

```
scaledAi = airy(0,x,1);
noscaleAi =  airy(0,x,0);
```

Plot the real part of each result.

```
rscaled = real(scaledAi);
rnoscale = real(noscaleAi);
figure
plot(x,rscaled,'-b',x,rnoscale,'-r');
axis([-10 1 -0.60 0.60]);
xlabel('x');
legend('scaled','not scaled','Location','SouthEast');
```



See Also

`besseli` | `besselj` | `besselk` | `bessely` | `besselh`

align

Purpose

Align user interface controls (`uicontrols`) and axes

Syntax

```
align(HandleList,'HorizontalAlignment','VerticalAlignment')
Positions = align(HandleList, 'HorizontalAlignment',
   'VerticalAlignment')
Positions = align(CurPositions, 'HorizontalAlignment',
   'VerticalAlignment')
```

Description

`align(HandleList,'HorizontalAlignment','VerticalAlignment')` aligns the `uicontrol` and `axes` objects in `HandleList`, a vector of handles, according to the options `HorizontalAlignment` and `VerticalAlignment`. The following tables show the possible values for `HorizontalAlignment` and `VerticalAlignment`.

HorizontalAlignment	Definition
None	No horizontal alignment
Left	Shifts the objects' left edges to that of the first object selected
Center	Shifts objects to center their positions to the average of the extreme *x*-values of the group
Right	Shifts the objects' right edges to that of the first object selected
Distribute	Equalizes *x*-distances between all objects within the span of the extreme *x*-values
Fixed	Spaces objects to have a specified number of points between them in the *y*-direction

VerticalAlignment	Definition
None	No vertical alignment
Top	Shifts the objects' top edges to that of the first object selected

VerticalAlignment	Definition
Middle	Shifts objects to center their positions to the average of the extreme *y*-values of the group
Bottom	Shifts the objects' bottom edges to that of the first object selected
Distribute	Equalizes *y*-distances between all objects within the span of the extreme *y*-values
Fixed	Spaces objects to have a specified number of points between them in the *x*-direction

Aligning objects does not change their absolute sizes. All alignment options align the objects within the bounding box that encloses the objects. `Distribute` and `Fixed` align objects to the bottom left of the bounding box. `Distribute` evenly distributes the objects while `Fixed` distributes the objects with a fixed distance (in points) between them. When you specify both horizontal and vertical distance together, the keywords `'HorizontalAlignment'` and `'VerticalAlignment'` are not necessary.

If you use `Fixed` for `HorizontalAlignment` or `VerticalAlignment`, you must also specify the distance, in points, where 72 points equals 1 inch. For example:

```
align(HandleList,'Fixed',Distance,'VerticalAlignment')
```

distributes the specified components `Distance` points horizontally and aligns them vertically as specified.

```
align(HandleList,'HorizontalAlignment','Fixed',Distance)
```

aligns the specified components horizontally as specified and distributes them `Distance` points vertically.

```
align(HandleList,'Fixed',HorizontalDistance,...
      'Fixed',VerticalDistance)
```

distributes the specified components `HorizontalDistance` points horizontally and distributes them `VerticalDistance` points vertically.

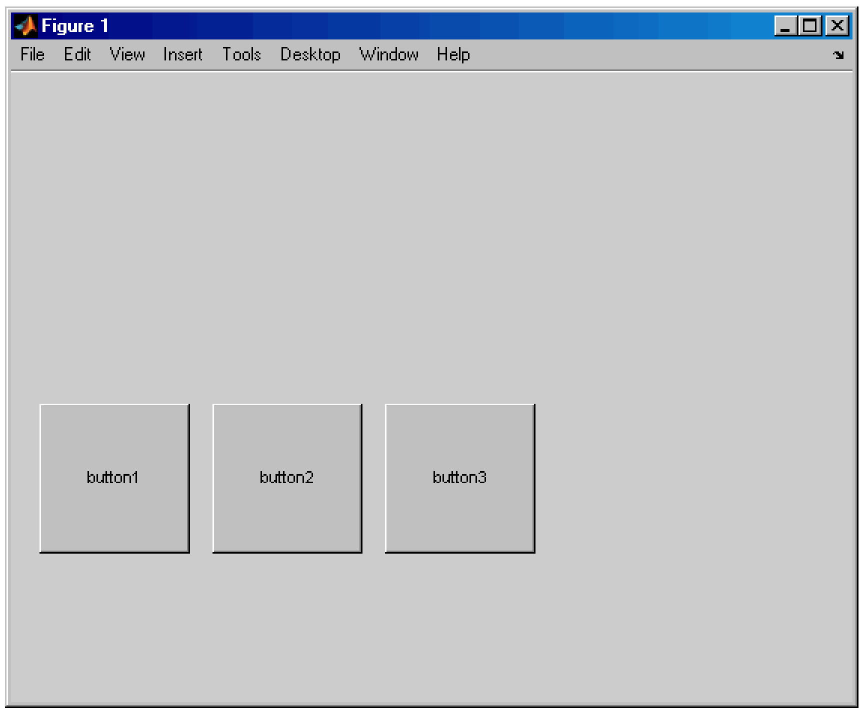
`Positions = align(HandleList, 'HorizontalAlignment', 'VerticalAlignment')` returns updated positions for the specified objects as a vector of `Position` vectors. The position of the objects on the figure does not change.

`Positions = align(CurPositions, 'HorizontalAlignment', 'VerticalAlignment')` returns updated positions for the objects whose positions are contained in `CurPositions`, where `CurPositions` is a vector of `Position` vectors. The position of the objects on the figure does not change.

Examples

Create a GUI with three buttons and use `align` to line up the buttons.

```
% Create a figure window and one button object:
f=figure;
u1 = uicontrol('Style','push', 'parent', f,'pos',...
[20 100 100 100],'string','button1');
% Create two more button objects, not aligned with
% each other or any part of the figure window:
u2 = uicontrol('Style','push', 'parent', f,'pos',...
[150 250 100 100],'string','button2');
u3 = uicontrol('Style','push', 'parent', f,'pos',...
[250 100 100 100],'string','button3');
% Align the button objects with the bottom of the first
% button object, equalizing the distance between the
% objects within the span of the extreme x-values:
align([u1 u2 u3],'distribute','bottom');
```



Alternatives

See “Alignment Tool — Aligning and Distributing Objects” for the GUI alternative.

See Also

`uicontrol` | `uistack`

alim

Purpose

Set or query axes alpha limits

Syntax

```
alpha_limits = alim
alim([amin amax])
alim_mode = alim('mode')
alim('alim_mode')
alim(axes_handle,...)
```

Description

`alpha_limits = alim` returns the alpha limits (`ALim` property) of the current axes.

`alim([amin amax])` sets the alpha limits to the specified values. `amin` is the value of the data mapped to the first alpha value in the alphamap, and `amax` is the value of the data mapped to the last alpha value in the alphamap. Data values in between are linearly interpolated across the alphamap, while data values outside are clamped to either the first or last alphamap value, whichever is closest.

`alim_mode = alim('mode')` returns the alpha limits mode (`ALimMode` property) of the current axes.

`alim('alim_mode')` sets the alpha limits mode on the current axes. `alim_mode` can be

- `auto` — MATLAB automatically sets the alpha limits based on the alpha data of the objects in the axes.
- `manual` — MATLAB does not change the alpha limits.

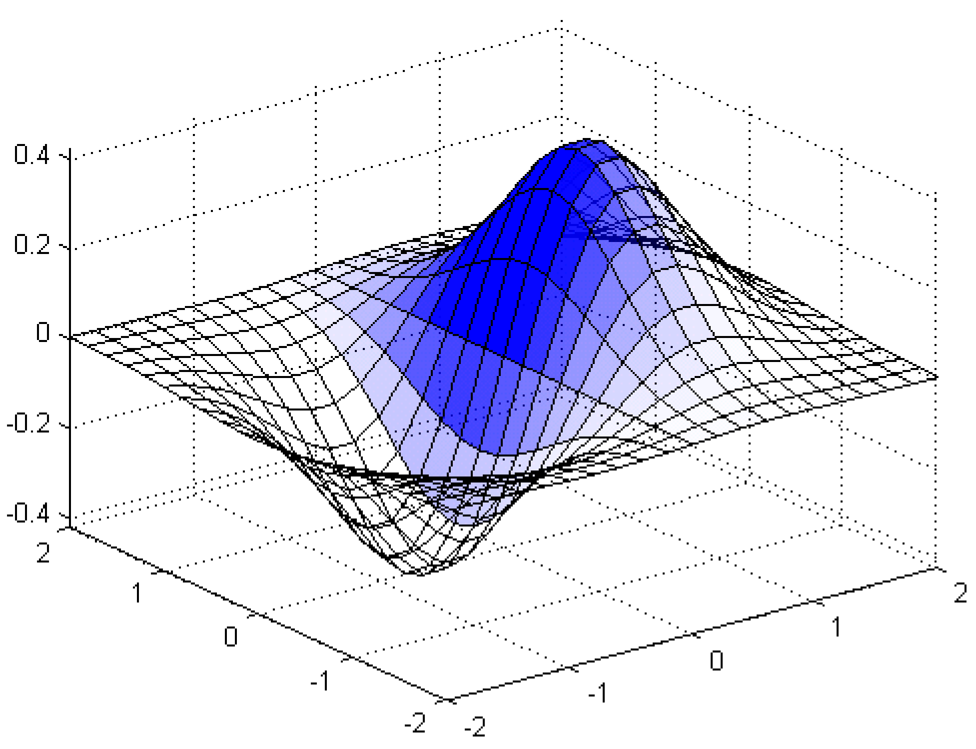
`alim(axes_handle,...)` operates on the specified axes.

Examples

Map transparency to a surface plot of `z`-data and change the `alim` property to make all values below zero transparent:

```
[x,y] = meshgrid([-2:.2:2]);
z = x.*exp(-x.^2-y.^2);
% Plot the data, using the gradient of z as
% the alphamap:
surf(x,y,z+.001,'FaceAlpha','flat',...
'AlphaDataMapping','scaled',...
```

```
'AlphaData',gradient(z),...
'FaceColor','blue');
axis tight
% Adjust the alim property to see only where
% the gradient is between 0 and 0.15:
alim([0 .15])
```



See Also

`alpha` | `alphamap` | `caxis` | Axes: `ALim` | Axes: `ALimMode` | Surface: `AlphaData` | Patch: `FaceVertexAlphaData`

- "Making Objects Transparent"

all

Purpose

Determine if all array elements are nonzero or `true`

Syntax

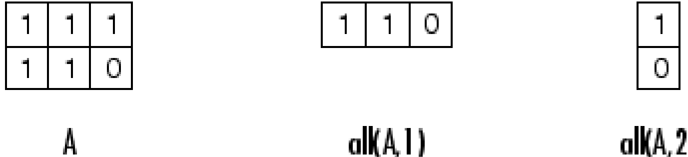
```
B = all(A)
B = all(A, dim)
```

Description

`B = all(A)` tests whether *all* the elements along various dimensions of an array are nonzero or logical `1` (`true`).

- If `A` is a vector, `all(A)` returns logical `1` (`true`) if all the elements are nonzero and returns logical `0` (`false`) if one or more elements are zero.
- If `A` is a nonempty matrix, `all(A)` treats the columns of `A` as vectors, returning a row vector of logical `1`'s and `0`'s.
- If `A` is an empty 0-by-0 matrix, `all(A)` returns logical `1` (`true`).
- If `A` is a multidimensional array, `all(A)` acts along the first nonsingleton dimension and returns an array of logical values. The size of this dimension reduces to `1` while the sizes of all other dimensions remain the same.

`B = all(A, dim)` tests along the dimension of `A` specified by scalar *`dim`*.



Examples

Given

```
A = [0.53 0.67 0.01 0.38 0.07 0.42 0.69]
```

then `B = (A < 0.5)` returns logical `1` (`true`) only where `A` is less than one half:

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

The all function reduces such a vector of logical conditions to a single condition. In this case, all(B) yields 0.

This makes all particularly useful in if statements:

```
if all(A < 0.5)
    do something
end
```

where code is executed depending on a single condition, not a vector of possibly conflicting conditions.

To determine if all elements in an entire array are nonzero, use all in conjunction with the colon operator.

```
A = eye(3);
all(A(:))

ans =

     0
```

Definitions

First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to 1.

For example:

- If X is a 1-by-n row vector, then the second dimension is the first nonsingleton dimension of X.
- If X is a 1-by-0-by-n empty array, then the second dimension is the first nonsingleton dimension of X.
- If X is a 1-by-1-by-3 array, then the third dimension is the first nonsingleton dimension of X.

See Also

any | colon (:) | max | mean | median | min | prod | std | sum | trapz | Logical Operators: Elementwise | Logical Operators: Short-circuit | Relational Operators

allchild

Purpose

Find all children of specified objects

Syntax

```
child_handles = allchild(handle_list)
```

Description

`child_handles = allchild(handle_list)` returns the list of all children (including ones with hidden handles) for each handle. If `handle_list` is a single element, `allchild` returns the output in a vector. If `handle_list` is a vector of handles, the output is a cell array.

Examples

Compare the results these two statements return:

```
axes
get(gca,'Children')
allchild(gca)
```

See Also

`findall` | `findobj`

Purpose

Set transparency properties for objects in current axes

Syntax

```
alpha
alpha(object_handle,value)
alpha(face_alpha)
alpha(alpha_data)
alpha(alpha_data)
alpha(alpha_data_mapping)
```

Description

`alpha` sets one of three transparency properties, depending on what arguments you specify with the call to this function. For available arguments, see Inputs.

`alpha(object_handle,value)` sets the transparency property only on the object identified by `object_handle`.

Input Arguments

Face Alpha

`alpha(face_alpha)` sets the `FaceAlpha` property of all image, patch, and surface objects in the current axes. You can set `face_alpha` to

scalar	Set the `FaceAlpha` property to the specified value (for images, set the `AlphaData` property to the specified value).
`'flat'`	Set the `FaceAlpha` property to `flat`.
`'interp'`	Set the `FaceAlpha` property to `interp`.
`'texture'`	Set the `FaceAlpha` property to `texture`.
`'opaque'`	Set the `FaceAlpha` property to `1`.
`'clear'`	Set the `FaceAlpha` property to `0`.

See “Specifying Transparency” for more information.

AlphaData (Surface Objects)

`alpha(alpha_data)` sets the `AlphaData` property of all surface objects in the current axes. You can set `alpha_data` to

matrix the same size as CData	Set the AlphaData property to the specified values.
'x'	Set the AlphaData property to be the same as XData.
'y'	Set the AlphaData property to be the same as YData.
'z'	Set the AlphaData property to be the same as ZData.
'color'	Set the AlphaData property to be the same as CData.
'rand'	Set the AlphaData property to a matrix of random values equal in size to CData.

AlphaData (Image Objects)

alpha(alpha_data) sets the AlphaData property of all image objects in the current axes. You can set alpha_data to

matrix the same size as CData	Set the AlphaData property to the specified value.
'x'	Ignored.
'y'	Ignored.
'z'	Ignored.
'color'	Set the AlphaData property to be the same as CData.
'rand'	Set the AlphaData property to a matrix of random values equal in size to CData.

AlphaDataMapping

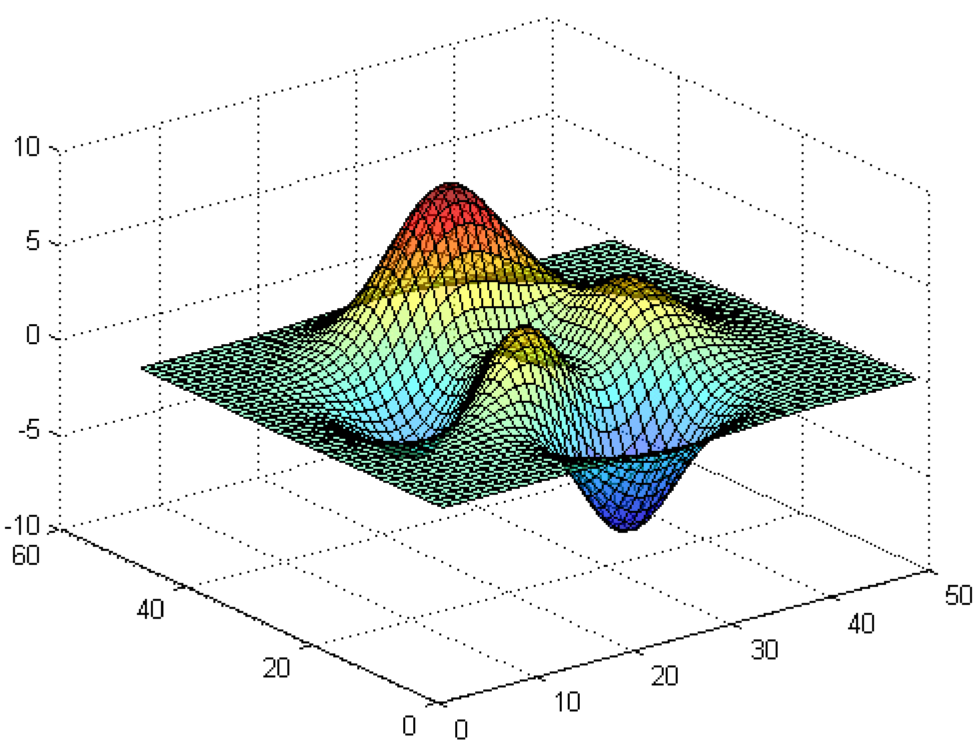
alpha(alpha_data_mapping) sets the AlphaDataMapping property of all image, patch, and surface objects in the current axes. You can set alpha_data_mapping to

'scaled'	Set the AlphaDataMapping property to scaled.
'direct'	Set the AlphaDataMapping property to direct.
'none'	Set the AlphaDataMapping property to none.

Examples

Create a surface plot and change its transparency using alpha:

```
surf(peaks);
alpha(0.5);
```



See Also

`alim` | `alphamap` | Image: `AlphaData` | Image: `AlphaDataMapping` | Patch: `FaceAlpha` | Patch: `FaceVertexAlphaData` | Patch: `AlphaDataMapping` | Surface: `FaceAlpha` | Surface: `AlphaData` | Surface: `AlphaDataMapping`

- "Making Objects Transparent"

Purpose

Specify figure alphamap (transparency)

Syntax

```
alphamap(alpha_map)
alphamap('parameter')
alphamap('parameter',length)
alphamap('parameter',delta)
alphamap(figure_handle,...)
alpha_map = alphamap
alpha_map = alphamap(figure_handle)
alpha_map = alphamap('parameter')
```

Description

`alphamap(alpha_map)` sets the `AlphaMap` of the current figure to the specified `m`-by-1 array of alpha values, `alpha_map`.

`alphamap('parameter')` creates a new alphamap or modifies the current alphamap. You can specify the following parameters:

- `'default'` — Set the `AlphaMap` property to the figure's default alphamap.
- `'rampup'` — Create a linear alphamap with increasing opacity (default `length` equals the current alphamap length).
- `'rampdown'` — Create a linear alphamap with decreasing opacity (default `length` equals the current alphamap length).
- `'vup'` — Create an alphamap that is opaque in the center and becomes more transparent linearly towards the beginning and end (default `length` equals the current alphamap length).
- `'vdown'` — Create an alphamap that is transparent in the center and becomes more opaque linearly towards the beginning and end (default `length` equals the current alphamap length).
- `'increase'` — Modify the alphamap making it more opaque (default `delta` is `.1`, added to the current values).
- `'decrease'` — Modify the alphamap making it more transparent (default `delta` is `.1`, subtracted from the current values).

- `'spin'` — Rotate the current alphamap (default `delta` is 1; `delta` must be an integer).

`alphamap('parameter',length)` creates a new alphamap with the length specified by the integer `length` (used with parameters `'rampup'`, `'rampdown'`, `'vup'`, `'vdown'`).

`alphamap('parameter',delta)` modifies the existing alphamap using the value specified by the integer `delta` (used with parameters `'increase'`, `'decrease'`, `'spin'`).

`alphamap(figure_handle,...)` performs the operation on the alphamap of the figure identified by `figure_handle`.

`alpha_map = alphamap` returns the current alphamap.

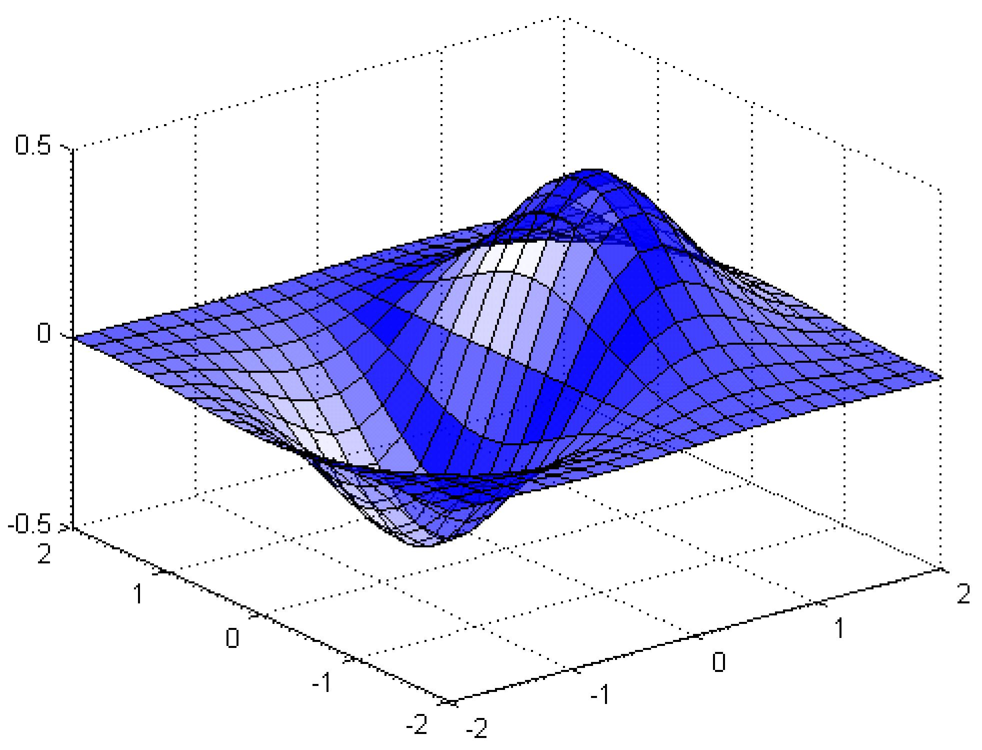
`alpha_map = alphamap(figure_handle)` returns the current alphamap from the figure identified by `figure_handle`.

`alpha_map = alphamap('parameter')` returns the alphamap modified by the `parameter`, but does not set the `AlphaMap` property.

Examples

Create a surface plot and change the alphamap

```
[x,y] = meshgrid([-2:.2:2]);
z = x.*exp(-x.^2-y.^2);
% Plot the data, using the gradient of z as
% the alphamap:
surf(x,y,z+.001,'FaceAlpha','flat',...
'AlphaDataMapping','scaled',...
'AlphaData',gradient(z),...
'FaceColor','blue');
% Change the alphamap to be opaque at the middle and
% transparent towards the ends:
alphamap('vup')
```



See Also

`alim` | `alpha` | Image: `AlphaData` | Image: `AlphaDataMapping` | Patch: `FaceAlpha` | Patch: `FaceVertexAlphaData` | Patch: `AlphaDataMapping` | Surface: `FaceAlpha` | Surface: `AlphaData` | Surface: `AlphaDataMapping`

- "Making Objects Transparent"

Purpose

Approximate minimum degree permutation

Syntax

```
P = amd(A)
P = amd(A,opts)
```

Description

`P = amd(A)` returns the approximate minimum degree permutation vector for the sparse matrix `C = A + A'`. The Cholesky factorization of `C(P,P)` or `A(P,P)` tends to be sparser than that of `C` or `A`. The `amd` function tends to be faster than `symamd`, and also tends to return better orderings than `symamd`. Matrix `A` must be square. If `A` is a full matrix, then `amd(A)` is equivalent to `amd(sparse(A))`.

`P = amd(A,opts)` allows additional options for the reordering. The `opts` input is a structure with the two fields shown below. You only need to set the fields of interest:

- **dense** — A nonnegative scalar value that indicates what is considered to be dense. If A is n-by-n, then rows and columns with more than `max(16,(dense*sqrt(n)))` entries in `A + A'` are considered to be "dense" and are ignored during the ordering. MATLAB software places these rows and columns last in the output permutation. The default value for this field is 10.0 if this option is not present.
- **aggressive** — A scalar value controlling aggressive absorption. If this field is set to a nonzero value, then aggressive absorption is performed. This is the default if this option is not present.

MATLAB software performs an assembly tree post-ordering, which is typically the same as an elimination tree post-ordering. It is not always identical because of the approximate degree update used, and because “dense” rows and columns do not take part in the post-order. It well-suited for a subsequent `chol` operation, however, If you require a precise elimination tree post-ordering, you can use the following code:

```
P = amd(S);
C = spones(S)+spones(S');
[ignore, Q] = etree(C(P,P));
P = P(Q);
```

If `S` is already symmetric, omit the second line, `C = spones(S)+spones(S')`.

Examples

This example constructs a sparse matrix and computes a two Cholesky factors: one of the original matrix and one of the original matrix preordered by `amd`. Note how much sparser the Cholesky factor of the preordered matrix is compared to the factor of the matrix in its natural ordering:

```
A = gallery('wathen',50,50);
p = amd(A);
L = chol(A,'lower');
Lp = chol(A(p,p),'lower');

figure;
subplot(2,2,1);    spy(A);
title('Sparsity structure of A');

subplot(2,2,2); spy(A(p,p));
title('Sparsity structure of AMD ordered A');

subplot(2,2,3); spy(L);
title('Sparsity structure of Cholesky factor of A');

subplot(2,2,4); spy(Lp);
title('Sparsity structure of Cholesky factor of AMD ordered A');

set(gcf,'Position',[100 100 800 700]);
```

See Also

`colamd` | `colperm` | `symamd` | `symrcm` | `Arithmetic Operator /`

ancestor

Purpose

Ancestor of graphics object

Syntax

```
p = ancestor(h,type)
p = ancestor(h,type,'toplevel')
```

Description

`p = ancestor(h,type)` returns the handle of the closest ancestor of `h`, if the ancestor is one of the types of graphics objects specified by `type`. `type` can be:

- a string that is the name of a single type of object. For example, `'figure'`
- a cell array containing the names of multiple objects. For example, `{'hgtransform','hggroup','axes'}`

If MATLAB cannot find an ancestor of `h` that is one of the specified types, then `ancestor` returns `p` as empty. When `ancestor` searches the hierarchy, it includes the object itself in the search. Therefore, if the object with handle `h` is of one of the types listed in `type`, `ancestor` will return object `h`.

`ancestor` returns `p` as empty but does not issue an error if `h` is not the handle of a Handle Graphics object.

`p = ancestor(h,type,'toplevel')` returns the highest-level ancestor of `h`, if this type appears in the `type` argument.

Examples

Find the ancestors of a line object:

```
% Create some line objects and parent them
% to an hggroup object.
hgg = hggroup;
hgl = line(randn(5),randn(5),'Parent',hgg);
% Now get the ancestor of the lines:
p = ancestor(hgg,{'figure','axes','hggroup'});
get(p,'Type')
% Now get the top-level ancestor:
ptop=ancestor(hgg,{'figure','axes','hggroup'},'toplevel');
get(ptop,'type')
```

See Also

`findobj`

Purpose

Find logical AND of array or scalar inputs

Syntax

```
A & B & ...
and(A, B)
```

Description

`A & B & ...` performs a logical AND of all input arrays `A`, `B`, etc., and returns an array containing elements set to either logical 1 (`true`) or logical 0 (`false`). An element of the output array is set to 1 if all input arrays contain a nonzero element at that same array location. Otherwise, that element is set to 0.

Each input of the expression can be an array or can be a scalar value. All nonscalar input arrays must have equal dimensions. If one or more inputs are an array, then the output is an array of the same dimensions. If all inputs are scalar, then the output is scalar.

If the expression contains both scalar and nonscalar inputs, then each scalar input is treated as if it were an array having the same dimensions as the other input arrays. In other words, if input `A` is a 3-by-5 matrix and input `B` is the number 1, then `B` is treated as if it were a 3-by-5 matrix of ones.

`and(A, B)` is called for the syntax `A & B` when either `A` or `B` is an object.

Note The symbols `&` and `&&` perform different operations in the MATLAB software. The element-wise AND operator described here is `&`. The short-circuit AND operator is `&&`.

Examples

If matrix `A` is

```
0.4235    0.5798         0    0.7942         0
0.5155         0    0.7833    0.0592    0.8744
0.3340         0         0         0    0.0150
0.4329    0.6405    0.6808    0.0503         0
```

and matrix `B` is

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

then

```
A & B
ans =
     0     1     0     1     0
     1     0     1     0     1
     0     0     0     0     0
     0     1     0     0     0
```

See Also

`bitand` | `or` | `xor` | `not` | `any` | `all`

How To

- logical operators
- bitwise functions

Purpose

Phase angle

Syntax

```
P = angle(Z)
```

Description

`P = angle(Z)` returns the phase angles, in radians, for each element of complex array `Z`. The angles lie between $\pm\pi$.

For complex `Z`, the magnitude `R` and phase angle `theta` are given by

```
R = abs(Z)
theta = angle(Z)
```

and the statement

```
Z = R.*exp(i*theta)
```

converts back to the original complex `Z`.

Examples

```
Z = [ 1 - 1i   2 + 1i   3 - 1i   4 + 1i
      1 + 2i   2 - 2i   3 + 2i   4 - 2i
      1 - 3i   2 + 3i   3 - 3i   4 + 3i
      1 + 4i   2 - 4i   3 + 4i   4 - 4i ]

P = angle(Z)

P =
   -0.7854    0.4636   -0.3218    0.2450
    1.1071   -0.7854    0.5880   -0.4636
   -1.2490    0.9828   -0.7854    0.6435
    1.3258   -1.1071    0.9273   -0.7854
```

Algorithms

The `angle` function can be expressed as `angle(z) = imag(log(z)) = atan2(imag(z),real(z))`.

See Also

`abs` | `atan2` | `unwrap`

Purpose

Create annotation objects

Syntax

```
annotation(annotation_type)
annotation('line',x,y)
annotation('arrow',x,y)
annotation('doublearrow',x,y)
annotation('textarrow',x,y)
annotation('textbox',[x y w h])
annotation('ellipse',[x y w h])
annotation('rectangle',[x y w h])
annotation(figure_handle,...)
annotation(...,'PropertyName',PropertyValue,...)
anno_obj_handle = annotation(...)
```

Description

`annotation(`*`annotation_type`*`)` creates the specified annotation type using default values for all properties. *`annotation_type`* can be one of the following strings:

- `'line'`
- `'arrow'`
- `'doublearrow'` (two-headed arrow),
- `'textarrow'` (arrow with attached text box),
- `'textbox'`
- `'ellipse'`
- `'rectangle'`

`annotation('line',x,y)` creates a line annotation object that extends from the point defined by `x(1)`,`y(1)` to the point defined by `x(2)`,`y(2)`, specified in normalized figure units.

`annotation('arrow',x,y)` creates an arrow annotation object that extends from the point defined by `x(1)`,`y(1)` to the point defined by `x(2)`,`y(2)`, specified in normalized figure units.

annotation('doublearrow',x,y) creates a two-headed annotation object that extends from the point defined by x(1),y(1) to the point defined by x(2),y(2), specified in normalized figure units.

annotation('textarrow',x,y) creates a textarrow annotation object that extends from the point defined by x(1),y(1) to the point defined by x(2),y(2), specified in normalized figure units. The tail end of the arrow is attached to an editable text box.

annotation('textbox',[x y w h]) creates an editable text box annotation with its lower left corner at the point x,y, a width w, and a height h, specified in normalized figure units. Specify x, y, w, and h in a single vector.

To type in the text box, enable plot edit mode (plotedit) and double-click within the box.

annotation('ellipse',[x y w h]) creates an ellipse annotation with the lower left corner of the bounding rectangle at the point x,y, a width w, and a height h, specified in normalized figure units. Specify x, y, w, and h in a single vector.

annotation('rectangle',[x y w h]) creates a rectangle annotation with the lower left corner of the rectangle at the point x,y, a width w, and a height h, specified in normalized figure units. Specify x, y, w, and h in a single vector.

annotation(figure_handle,...) creates the annotation in the specified figure.

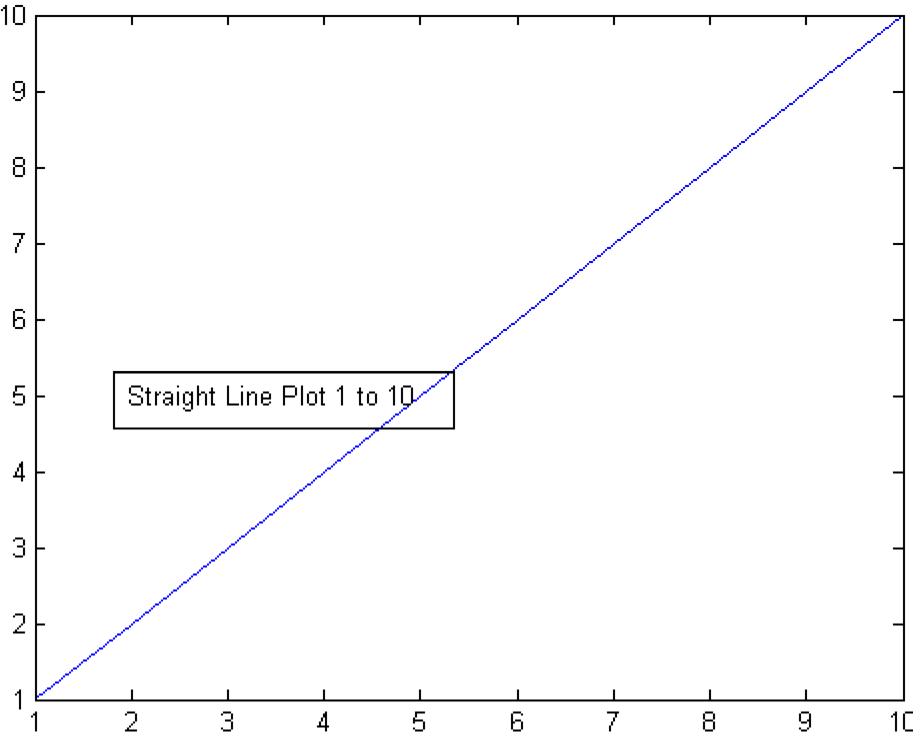
annotation(...,'PropertyName',PropertyValue,...) creates the annotation and sets the specified properties to the specified values.

anno_obj_handle = annotation(...) returns the handle to the "Annotation Objects".

Examples

Annotate a graph with a text box.

```
plot(1:10);
annotation('textbox', [.2 .4 .1 .1], 'String', 'Straight Line Plot 1 to 1
```

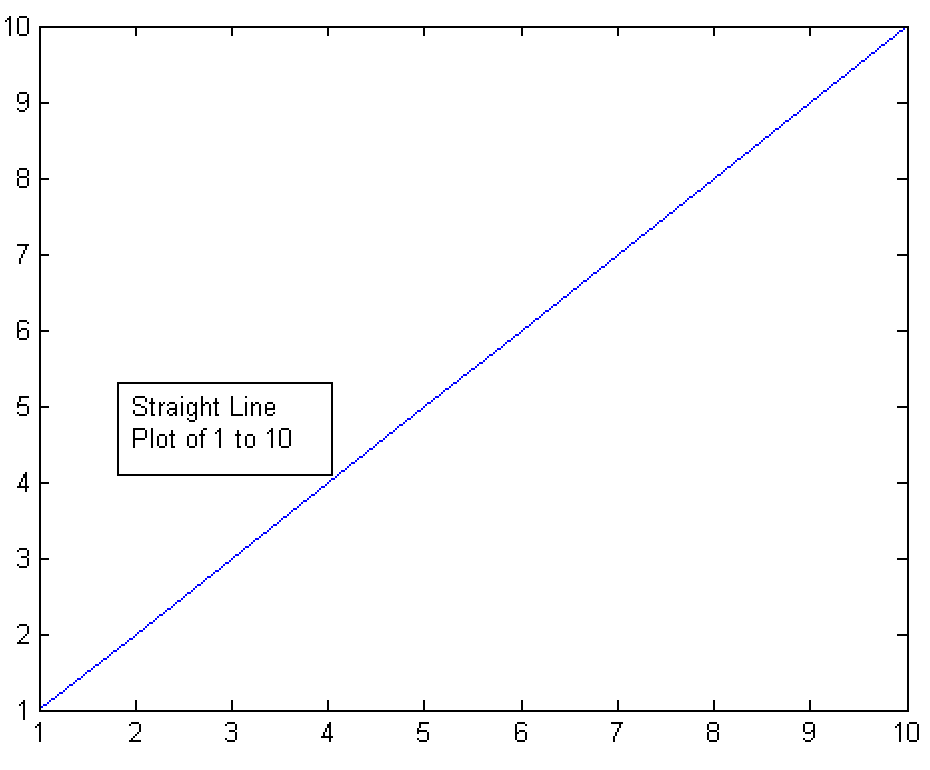


Notice that the locations specified are normalized in the figure coordinates. Length of the string automatically adjusts the width of the text box, as, `FitBoxToText` property of the annotation object is `on` by default.

Insert multiple lines in a text box by creating a cell array of string. Each element is used as a separate line.

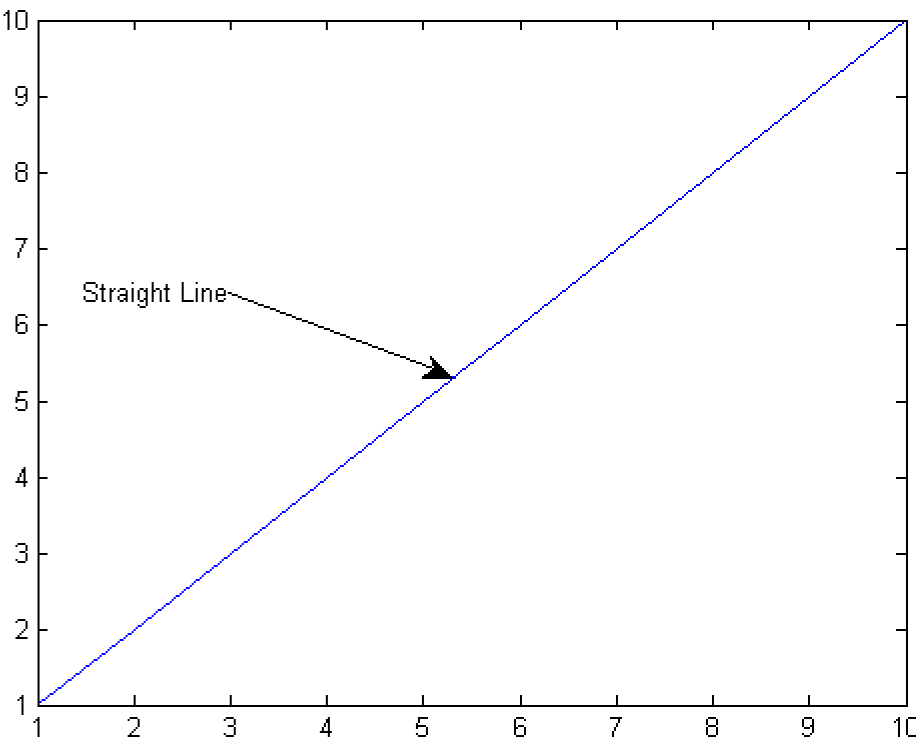
```
plot(1:10);
str = {'Straight Line', 'Plot of 1 to 10'};
```

```
annotation('textbox', [.2 .4, .1, .1], 'String', str);
```



Insert a text arrow in a graph.

```
plot(1:10);
a = annotation('textarrow', [.3 .5], [.6 .5], 'String' , 'Straight Line');
```



The text arrow is defined by points starting from `(.3, .6)` extending to `(.5 ,.5)` in normalized figure coordinates.

Adding Annotations Interactively

It is often convenient to place annotations interactively. For details, see “Working in Plot Edit Mode”.

See Also

`Annotation Arrow Properties` | `Annotation Doublearrow Properties` | `Annotation Ellipse Properties` | `Annotation Line`

`Properties` | `Annotation Rectangle Properties` | `Annotation Textarrow Properties` | `Annotation Textbox Properties`

How To

- "How to Annotate Graphs"
- "Annotation Objects"

Purpose

Define annotation arrow properties

Modifying Properties

You can set and query annotation object properties using the `set` and `get` functions and the Property Editor (displayed with the `propertyeditor` command).

Use the `annotation` function to create annotation objects and obtain their handles. For an example of its use, see "Positioning Annotations in Data Space" in the MATLAB Graphics documentation.

Annotation Arrow Property Descriptions

`Color`
ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object's color. The default value is `[0 0 0]` (`black`).

See the `ColorSpec` reference page for more information on specifying color. See "Adding Arrows and Lines to Graphs".

`HeadLength`
size in points

Length of the arrowhead. Specify this property in points. `1` point = $^{1}/_{72}$ inch. The default value is `10`. See also `HeadWidth`.

`HeadStyle`
string

Style of arrowhead. Specify this property as one of the strings from the following table.

Arrow Head Style Table

Head Style String	Head	Head Style String	Head
none		star4	
plain		rectangle	
ellipse		diamond	
vback1		rose	
vback2 (Default)		hypocycloid	
vback3		astroid	
cback1		deltoid	
cback2			
cback3			

`HeadWidth`
size in points

Width of arrowhead. Specify in points. 1 point = $^1/_{72}$ inch. The default value is 10. See also `HeadLength`.

`LineStyle`
`{-} | -- | : | -. | none`

Style of arrow stem.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line
'none'	No line

`LineWidth`
size in points

Width of arrow stem. Specify in points. 1 point = $^1/_{72}$ inch. The default is `0.5` points.

`Position`
four-element vector `[x, y, width, height]`

Size and location of the object. Specify the lower left corner of the object with the first two elements of the vector defining the point *x*, *y* in units normalized to the figure (when `Units` property is `normalized`). `width` and `height` specify the object's *dx* and *dy*, respectively, in units normalized to the figure. The default value is `[0.3 0.3 0.1 0.1]`.

`Units`
`{normalized} | inches | centimeters | characters | points | pixels`

Position units. MATLAB uses this property to determine the units used by the `Position` property. All positions are measured from the lower left corner of the figure window.

- `normalized` — Interpret `Position` as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly.
- `pixels`, `inches`, `centimeters`, and `points` — Absolute units. 1 point = $^1/_{72}$ inch.
- `characters` — Based on the size of characters in the default system font. The width of one `characters` unit is the width of the letter `x`, and the height of one `characters` unit is the distance between the baselines of two lines of text.

`X`

vector [X_{begin} X_{end}]

x-coordinates of beginning and ending points for line. A vector of *x*-axis (horizontal) values specifying the beginning and ending points of the line, units normalized to the figure. The default value is `[0.3 0.4]`.

`Y`

vector [Y_{begin} Y_{end}]

y-coordinates of beginning and ending points for line. A vector of *y*-axis (vertical) values specifying the beginning and ending points of the line, units normalized to the figure. The default value is `[0.3 0.4]`.

See Also

`annotation`

How To

- "Annotation Objects"
- "Adding Arrows and Lines to Graphs"

Purpose

Define annotation doublearrow properties

Modifying Properties

You can set and query annotation object properties using the `set` and `get` functions and the Property Editor (displayed with the `propertyeditor` command).

Use the `annotation` function to create annotation objects and obtain their handles. For an example of its use, see “Positioning Annotations in Data Space” in the MATLAB Graphics documentation.

Annotation Doublearrow Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`Color`
ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object’s color. The default value is `[0 0 0]` (`black`).

See the `ColorSpec` reference page for more information on specifying color. See “Adding Arrows and Lines to Graphs”.

`Head1Length`
size in points

Length of first arrowhead. Specify in points.`1` point = $^{1}/_{72}$ inch. The default value is `10`.

The first arrowhead is located at the end defined by the point `x(1), y(1)`.

See also the `Head1Style`, `Head1Width`, `X` and `Y` properties.

`Head2Length`
size in points

Length of second arrowhead. Specify in points.1 point = $^{1}/_{72}$ inch. The default value is 10.

The first arrowhead is located at the end defined by the point x(end), y(end).

See also the Head1Style, Head1Width, X and Y properties.

Head1Style
string

Style of first arrowhead. Specify this property as one of the strings from the following table.

Arrow Head Style Table

Head Style String	Head	Head Style String	Head
none		star4	
plain		rectangle	
ellipse		diamond	
vback1		rose	
vback2 (Default)		hypocycloid	
vback3		astroid	

Head Style String	Head	Head Style String	Head
cback1		deltoid	
cback2			
cback3			

See also `Head1Width`, `Head1Length`, and `LineWidth`.

`Head2Style`
string

Style of second arrowhead. Specify this property as one of the strings shown in the table for the `Head1Style` property.

See also `Head2Width`, `Head2Length`, `LineWidth`.

`Head1Width`
size in points

Width of first arrowhead. Specify in points.1 point = $^1/_{72}$ inch. See also `Head1Length`.

`Head2Width`
size in points

Width of second arrowhead. Specify in points.1 point = $^1/_{72}$ inch. See also `Head2Length`.

`LineStyle`
`{-} | -- | : | -. | none`

Style of arrow stem.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line
'none'	No line

See also `Head1Style`, `Head1Length`, and `LineWidth`.

`LineWidth`
size in points

Width of arrow stem. Specify in points.1 point = $^1/_{72}$ inch. The default is `0.5` points.

`Position`
four-element vector `[x, y, width, height]`

Size and location of the object. Specify the lower left corner of the object with the first two elements of the vector defining the point *x*, *y* in units normalized to the figure (when `Units` property is `normalized`). `width` and `height` specify the object's *dx* and *dy*, respectively, in units normalized to the figure. The default value is `[0.3 0.3 0.1 0.1]`.

`Units`
`{normalized} | inches | centimeters | characters | points | pixels`

Position units. MATLAB uses this property to determine the units used by the `Position` property. All positions are measured from the lower left corner of the figure window.

- `normalized` — Interpret `Position` as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly.
- `pixels`, `inches`, `centimeters`, and `points` — Absolute units. `1` point = $^{1}/_{72}$ inch.
- `characters` — Based on the size of characters in the default system font. The width of one `characters` unit is the width of the letter `x`, and the height of one `characters` unit is the distance between the baselines of two lines of text.

`X`

vector [X_{begin} X_{end}]

x-coordinates of beginning and ending points for line. A vector of *x*-axis (horizontal) values specifying the beginning and ending points of the line, units normalized to the figure. The default value is `[0.3 0.4]`.

`Y`

vector [Y_{begin} Y_{end}]

y-coordinates of beginning and ending points for line. A vector of *y*-axis (vertical) values specifying the beginning and ending points of the line, units normalized to the figure. The default value is `[0.3 0.4]`.

See Also

`annotation`

How To

- "Annotation Objects"
- "Adding Arrows and Lines to Graphs"

Annotation Ellipse Properties

Purpose

Define annotation ellipse properties

Modifying Properties

You can set and query annotation object properties using the `set` and `get` functions and the Property Editor (displayed with the `propertyeditor` command).

Use the `annotation` function to create annotation objects and obtain their handles. For an example of its use, see “Positioning Annotations in Data Space” in the MATLAB Graphics documentation.

Annotation Ellipse Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`EdgeColor`

ColorSpec | `none`

Edge color of object'. A three-element RGB vector or one of the MATLAB predefined names, specifying the edge color. The default value is `[0 0 0]` (`black`).

See the `ColorSpec` reference page for more information on specifying color.

`FaceColor`

`{flat} | none | ColorSpec`

Color of filled areas.

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for all filled areas.
- `none` — Do not draw faces. Note that MATLAB draws `EdgeColor` independently of `FaceColor`.
- `flat` — The object uses the figure `colormap` to determine the color of the filled areas.

`LineStyle`

`{-} | -- | : | -. | none`

Line style of ellipse.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line
'none'	No line

LineWidth
size in points

Width of linear objects and edges of filled areas. Specify in points. 1 point = $^1/_{72}$ inch. The default is 0.5 points.

Position
four-element vector [x, y, width, height]

Size and location of the object. Specify the lower left corner of the object with the first two elements of the vector defining the point *x*, *y* in units normalized to the figure (when Units property is normalized). width and height specify the object's *dx* and *dy*, respectively, in units normalized to the figure. The default value is [0.3 0.3 0.1 0.1].

Units
{normalized} | inches | centimeters | characters | points | pixels

Position units. MATLAB uses this property to determine the units used by the Position property. All positions are measured from the lower left corner of the figure window.

- `normalized` — Interpret `Position` as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly.
- `pixels`, `inches`, `centimeters`, and `points` — Absolute units. 1 point = $^1/_{72}$ inch.
- `characters` — Based on the size of characters in the default system font. The width of one `characters` unit is the width of the letter `x`, and the height of one `characters` unit is the distance between the baselines of two lines of text.

See Also

`annotation`

How To

- "Annotation Objects"
- "How to Annotate Graphs"

Purpose

Define annotation line properties

Modifying Properties

You can set and query annotation object properties using the `set` and `get` functions and the Property Editor (displayed with the `propertyeditor` command).

Use the `annotation` function to create annotation objects and obtain their handles. For an example of its use, see "Positioning Annotations in Data Space" in the MATLAB Graphics documentation.

Annotation Line Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`Color`

ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object's color. The default value is `[0 0 0]` (`black`).

See the `ColorSpec` reference page for more information on specifying color. See "Adding Arrows and Lines to Graphs".

`LineStyle`

`{-} | -- | : | -. | none`

Line style of annotation line object.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line

Specifier	Line Style
'-.'	Dash-dot line
'none'	No line

`LineWidth`
size in points

Width of linear objects and edges of filled areas. Specify in points. 1 point = $^1/_{72}$ inch. The default is `0.5` points.

`Position`
four-element vector `[x, y, width, height]`

Size and location of the object. Specify the lower left corner of the object with the first two elements of the vector defining the point *x*, *y* in units normalized to the figure (when `Units` property is `normalized`). `width` and `height` specify the object's *dx* and *dy*, respectively, in units normalized to the figure. The default value is `[0.3 0.3 0.1 0.1]`.

`Units`
`{normalized} | inches | centimeters | characters | points | pixels`

Position units. MATLAB uses this property to determine the units used by the `Position` property. All positions are measured from the lower left corner of the figure window.

- `normalized` — Interpret `Position` as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly.
- `pixels`, `inches`, `centimeters`, and `points` — Absolute units. 1 point = $^1/_{72}$ inch.
- `characters` — Based on the size of characters in the default system font. The width of one `characters` unit is the width

of the letter `x`, and the height of one `characters` unit is the distance between the baselines of two lines of text.

`X`

vector [X_{begin} X_{end}]

x-coordinates of beginning and ending points for line. A vector of *x*-axis (horizontal) values specifying the beginning and ending points of the line, units normalized to the figure. The default value is `[0.3 0.4]`.

`Y`

vector [Y_{begin} Y_{end}]

y-coordinates of beginning and ending points for line. A vector of *y*-axis (vertical) values specifying the beginning and ending points of the line, units normalized to the figure. The default value is `[0.3 0.4]`.

See Also

`annotation`

How To

- "Annotation Objects"
- "Adding Arrows and Lines to Graphs"

Annotation Rectangle Properties

Purpose

Define annotation rectangle properties

Modifying Properties

You can set and query annotation object properties using the `set` and `get` functions and the Property Editor (displayed with the `propertyeditor` command).

Use the `annotation` function to create annotation objects and obtain their handles. For an example of its use, see "Positioning Annotations in Data Space" in the MATLAB Graphics documentation.

Annotation Rectangle Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`EdgeColor`

ColorSpec | `none`

Edge color of object'. A three-element RGB vector or one of the MATLAB predefined names, specifying the edge color. The default value is `[0 0 0]` (`black`).

See the `ColorSpec` reference page for more information on specifying color.

`FaceAlpha`

Scalar alpha value in range `[0 1]`

Transparency of object background. Defines the degree to which the object's background color is transparent. A value of `1` (the default) makes the background opaque, a value of `0` makes the background completely transparent (that is, invisible). The default value is `1`.

`FaceColor`

`{flat} | none | ColorSpec`

Color of filled areas.

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for all filled areas.
- `none` — Do not draw faces. Note that MATLAB draws `EdgeColor` independently of `FaceColor`.
- `flat` — The object uses the figure `colormap` to determine the color of the filled areas.

`LineStyle`
`{-} | -- | : | -. | none`

Line style of annotation rectangle object.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line
'none'	No line

`LineWidth`
size in points

Width of linear objects and edges of filled areas. Specify in points. 1 point = $^1/_{72}$ inch. The default is `0.5` points.

`Position`
four-element vector `[x, y, width, height]`

Size and location of the object. Specify the lower left corner of the object with the first two elements of the vector defining the point *x*, *y* in units normalized to the figure (when `Units` property is `normalized`). `width` and `height` specify the object's *dx* and *dy*,

respectively, in units normalized to the figure. The default value is `[0.3 0.3 0.1 0.1]`.

`Units`

`{normalized} | inches | centimeters | characters | points | pixels`

Position units. MATLAB uses this property to determine the units used by the `Position` property. All positions are measured from the lower left corner of the figure window.

- `normalized` — Interpret `Position` as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly.
- `pixels`, `inches`, `centimeters`, and `points` — Absolute units. `1` point = $^1/_{72}$ inch.
- `characters` — Based on the size of characters in the default system font. The width of one `characters` unit is the width of the letter `x`, and the height of one `characters` unit is the distance between the baselines of two lines of text.

See Also

`annotation`

Purpose

Define annotation textarrow properties

Modifying Properties

You can set and query annotation object properties using the `set` and `get` functions and the Property Editor (displayed with the `propertyeditor` command).

Use the `annotation` function to create annotation objects and obtain their handles. For an example of its use, see "Positioning Annotations in Data Space" in the MATLAB Graphics documentation.

Annotation Textarrow Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`Color`

ColorSpec

Color of the arrow, text and text border. A three-element RGB vector or one of the MATLAB predefined names, specifying the color of the arrow, the color of the text (`TextColor` property), and the rectangle enclosing the text (`TextEdgeColor` property). The default value is `[0 0 0]` (`black`).

Setting the `Color` property also sets the `TextColor` and `TextEdgeColor` properties to the same color. However, if the value of the `TextEdgeColor` is `none`, it remains `none` and the text box is not displayed. You can set `TextColor` or `TextEdgeColor` independently without affecting other properties.

For example, if you want to create a textarrow with a red arrow and black text in a black box, you must:

1. Set the `Color` property to red — `set(h,'Color','r')`
2. Set the `TextColor` to black — `set(h,'TextColor','k')`
3. Set the `TextEdgeColor` to black .— `set(h,'TextEdgeColor','k')`

If you do not want display the text box, set the `TextEdgeColor` to `none`.

See the `ColorSpec` reference page for more information on specifying color.

`FontAngle`

`{normal} | italic | oblique`

Character slant. MATLAB uses this property to select a font from those available on your particular system. Generally, setting this property to `italic` or `oblique` selects a slanted font.

`FontName`

string

Font family. A string specifying the name of the font to use for the text. To display and print properly, this font must be supported on your system. The default font is `Helvetica`.

`FontSize`

size in points

Approximate size of text characters. Specify in points. 1 point = $^{1}/_{72}$ inch. The default size is `10`.

`FontUnits`

`{points} | normalized | inches | centimeters | pixels`

Font size units. MATLAB uses this property to determine the units used by the `FontSize` property. Normalized units interpret `FontSize` as a fraction of the height of the parent axes. When you resize the axes, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units. 1 point = $^{1}/_{72}$ inch.

`FontWeight`

`{normal} | bold | light | demi`

Weight of text characters. MATLAB uses this property to select a font from those available on your system. Generally, setting this property to bold or demi causes MATLAB to use a bold font.

HeadLength
size in points

Length of the arrowhead. Specify this property in points. 1 point = $^1/_{72}$ inch. The default value is 10. See also HeadWidth.

HeadStyle
string

Style of arrowhead. Specify this property as one of the strings from the following table.

Arrow Head Style Table

Head Style String	Head	Head Style String	Head
none		star4	
plain		rectangle	
ellipse		diamond	
vback1		rose	
vback2 (Default)		hypocycloid	
vback3		astroid	

Head Style String	Head	Head Style String	Head
cback1		deltoid	
cback2			
cback3			

HeadWidth
size in points

Width of arrowhead. Specify in points. 1 point = $^1/_{72}$ inch. The default value is 10. See also HeadLength.

HorizontalAlignment
{left} | center | right

Horizontal alignment of text. Specifies the horizontal justification of the text string. It determines where MATLAB places the string horizontally with regard to the points specified by the Position property.

Interpreter
latex | {tex} | none

Interpret TeX instructions. This property controls whether MATLAB interprets certain characters in the String property as TeX instructions (default) or displays all characters literally. The options are:

- latex — Supports a basic subset of the LaTeX markup language.
- tex — Supports a subset of plain TeX markup language. See the String property for a list of supported TeX instructions.

- none — Displays literal characters.

`LineStyle`
`{-} | -- | : | -. | none`

Line style of arrow stem.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line
'none'	No line

`LineWidth`
size in points

Width of arrow stem. Specify in points 1 point = $^{1}/_{72}$ inch. The default is `0.5` points.

`Position`
four-element vector `[x, y, width, height]`

Size and location of the object. Specify the lower left corner of the object with the first two elements of the vector defining the point *x*, *y* in units normalized to the figure (when `Units` property is `normalized`). `width` and `height` specify the object's *dx* and *dy*, respectively, in units normalized to the figure. The default value is `[0.3 0.3 0.1 0.1]`.

`String`
string

Text string. Specify this property as a quoted string for single-line strings, or as a cell array of strings, or a padded string matrix for multiline strings. MATLAB displays this string at the specified location. Vertical slash characters are not interpreted as line breaks in text strings, and are drawn as part of the text string. See Mathematical Symbols, Greek Letters, and TeX Characters for an example.

Note The words `default`, `factory`, and `remove` are reserved words that will not appear in a figure when quoted as a normal string. In order to display any of these words individually, type `'\reserved_word'` instead of `'reserved_word'`.

When the text `Interpreter` property is `tex` (the default), you can use a subset of TeX commands embedded in the string to produce special characters such as Greek letters and mathematical symbols. The following table lists these characters and the character sequences used to define them.

TeX Character Sequence Table

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\alpha	α	\upsilon	υ	\sim	~
\angle	∠	\phi	Φ	\leq	≤
\ast	*	\chi	χ	\infty	∞
\beta	β	\psi	ψ	\clubsuit	♣
\gamma	γ	\omega	ω	\diamondsuit	♦
\delta	δ	\Gamma	Γ	\heartsuit	♥
\epsilon	ε	\Delta	Δ	\spadesuit	♠

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	Θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta		\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon		\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	э	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in		\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	ο

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨			\copyright	©
\langle	〈				

You can also specify stream modifiers that control font type and color. The first four modifiers are mutually exclusive. However, you can use `\fontname` in combination with one of the other modifiers.

`TextBackgroundColor`

`ColorSpec | {none}`

Color of text background rectangle. A three-element RGB vector or one of the MATLAB predefined names, specifying the arrow color.

See the `ColorSpec` reference page for more information on specifying color.

`TextColor`

`ColorSpec`

Color of text. A three-element RGB vector or one of the MATLAB predefined names, specifying the arrow color. The default value is `[0 0 0]` (black).

See the `ColorSpec` reference page for more information on specifying color. Setting the `Color` property also sets this property.

`TextEdgeColor`
`ColorSpec | {none}`

Color of edge of text rectangle. A three-element RGB vector or one of the MATLAB predefined names, specifying the color of the rectangle that encloses the text.

See the `ColorSpec` reference page for more information on specifying color. Setting the `Color` property also sets this property.

`TextLineWidth`
width in points

Width of text rectangle edge. Specify in points. `1` point = $^1/_{72}$ inch. The default value is `0.5`.

`TextMargin`
size in pixels

Space around text. A value in pixels that defines the space around the text string, but within the rectangle. Default value is `5` pixels.

`TextRotation`
rotation angle in degrees

Text orientation. Determines the orientation of the text string. Specify values of rotation in degrees (positive angles cause counterclockwise rotation). Angles are absolute and not relative to previous rotations; a rotation of `0` degrees is always horizontal. Default is `0`.

`Units`
`{normalized} | inches | centimeters | characters | points | pixels`

Position units. MATLAB uses this property to determine the units used by the `Position` property. All positions are measured from the lower left corner of the figure window.

- `normalized` — Interpret `Position` as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly.
- `pixels`, `inches`, `centimeters`, and `points` — Absolute units. `1` point = $^1/_{72}$ inch.
- `characters` — Based on the size of characters in the default system font. The width of one `characters` unit is the width of the letter `x`, and the height of one `characters` unit is the distance between the baselines of two lines of text.

`VerticalAlignment`

`top | cap | {middle} | baseline | bottom`

Vertical alignment of text. Specifies the vertical justification of the text string. It determines where MATLAB places the string vertically with regard to the points specified by the `Position` property.

Note that `top` and `cap` both place the text at the top, while `baseline` and `bottom` both align the text on the bottom.

`X`

vector [X_{begin} X_{end}]

x-coordinates of beginning and ending points for line. A vector of *x*-axis (horizontal) values specifying the beginning and ending points of the line, units normalized to the figure. The default value is `[0.3 0.4]`.

`Y`

vector [Y_{begin} Y_{end}]

y-coordinates of beginning and ending points for line. A vector of *y*-axis (vertical) values specifying the beginning and ending points of the line, units normalized to the figure. The default value is `[0.3 0.4]`.

See Also

`annotation`

Annotation Textbox Properties

Purpose

Define annotation textbox properties

Modifying Properties

You can set and query annotation object properties using the `set` and `get` functions and the Property Editor (displayed with the `propertyeditor` command).

Use the `annotation` function to create annotation objects and obtain their handles. For an example of its use, see “Positioning Annotations in Data Space” in the MATLAB Graphics documentation.

Annotation Textbox Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`BackgroundColor`
    `ColorSpec | {none}`

Color of text background rectangle. A three-element RGB vector or one of the MATLAB predefined names, specifying the rectangle background color. The default value is `none`.

See the `ColorSpec` reference page for more information on specifying color.

`Color`
    `ColorSpec`

Text color. A three-element RGB vector or one of the predefined names, specifying the text color. The default value is black. See `ColorSpec` for more information on specifying color.

`EdgeColor`
    `ColorSpec | {none}`

Color of edge of text rectangle. A three-element RGB vector or one of the MATLAB predefined names, specifying the color of the rectangle that encloses the text.

See the `ColorSpec` reference page for more information on specifying color. Setting the `Color` property also sets this property.

`FaceAlpha`
Scalar alpha value in range `[0 1]`

Transparency of object background. Defines the degree to which the object's background color is transparent. A value of `1` (the default) makes the background opaque, a value of `0` makes the background completely transparent (that is, invisible). The default value is `1`.

`FitBoxToText`
`{on} | off`

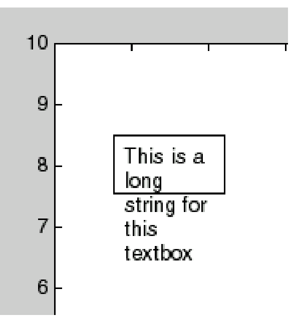
Automatically adjust text box width and height to fit text. When this property is `on` (the default), MATLAB automatically resizes textboxes to fit the *x*-extents and *y*-extents of the text strings they contain. When it is `off`, text strings are wrapped to fit the width of their textboxes, which can cause them to extend below the bottom of the box.

If you resize a textbox in plot edit mode or change the width or height of its `Position` property directly, MATLAB sets the object's `FitBoxToText` property to `off`. You can toggle this property with `set`, with the Property Inspector, or in plot edit mode via the object's context menu.

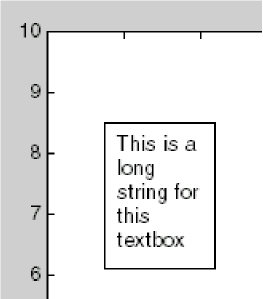
`FitHeightToText`
`on | off`

Automatically adjust text box width and height to fit text. MATLAB automatically wraps text strings to fit the width of the text box. However, if the text string is long enough, it can extend beyond the bottom of the text box.

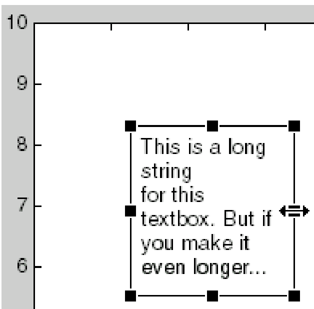
Note The property is obsolete. To control line wrapping behavior in textboxes, use `FitBoxToText` instead.



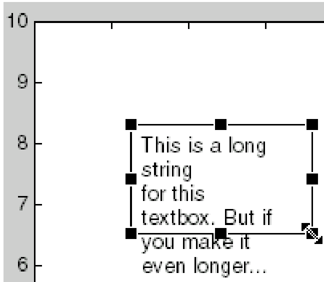
When you set this mode to `on`, MATLAB automatically adjusts the height of the text box to accommodate the string, doing so as you create or edit the string.



The fit-size-to-text behavior turns off if you resize the text box programmatically or manually in plot edit mode.



However, if you resize the text box from any other handles, the position you set is honored without regard to how the text fits the box.



`FontAngle`

`{normal} | italic | oblique`

Character slant. MATLAB uses this property to select a font from those available on your particular system. Generally, setting this property to `italic` or `oblique` selects a slanted font.

`FontName`

string

Font family. A string specifying the name of the font to use for the text. To display and print properly, this font must be supported on your system. The default font is `Helvetica`.

FontSize
size in points

Approximate size of text characters. Specify in points. 1 point = $^{1}/_{72}$ inch. The default size is 10.

FontUnits
{points} | normalized | inches | centimeters | pixels

Font size units. MATLAB uses this property to determine the units used by the FontSize property. Normalized units interpret FontSize as a fraction of the height of the parent axes. When you resize the axes, MATLAB modifies the screen FontSize accordingly. pixels, inches, centimeters, and points are absolute units. 1 point = $^{1}/_{72}$ inch.

FontWeight
{normal} | bold | light | demi

Weight of text characters. MATLAB uses this property to select a font from those available on your system. Generally, setting this property to bold or demi causes MATLAB to use a bold font.

HorizontalAlignment
{left} | center | right

Horizontal alignment of text. Specifies the horizontal justification of the text string within the textbox. It determines where MATLAB places the string horizontally with regard to the points specified by the Position property.

Interpreter
latex | {tex} | none

Interpret TeX instructions. This property controls whether MATLAB interprets certain characters in the String property as TeX instructions (default) or displays all characters literally. The options are:

- `latex` — Supports a basic subset of the LaTeX markup language.
- `tex` — Supports a subset of plain TeX markup language. See the `String` property for a list of supported TeX instructions.
- `none` — Displays literal characters.

`LineStyle`

`{-} | -- | :  | -.  | none`

Line style of annotation textbox object.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line
'none'	No line

`LineWidth`

size in points

Width of linear objects and edges of filled areas. Specify in points. 1 point = $^1/_{72}$ inch. The default is `0.5` points.

`Margin`

size in pixels

Space around text. A value in pixels that defines the space around the text string, but within the rectangle. Default value is `5` pixels.

`Position`

four-element vector `[x, y, width, height]`

Size and location of the object. Specify the lower left corner of the object with the first two elements of the vector defining the point *x*, *y* in units normalized to the figure (when `Units` property is `normalized`). `width` and `height` specify the object's *dx* and *dy*, respectively, in units normalized to the figure. The default value is `[0.3 0.3 0.1 0.1]`.

`String`
string

Text string. Specify this property as a quoted string for single-line strings, or as a cell array of strings, or a padded string matrix for multiline strings. MATLAB displays this string at the specified location. Vertical slash characters are not interpreted as line breaks in text strings, and are drawn as part of the text string. See Mathematical Symbols, Greek Letters, and TeX Characters for an example.

Note The words `default`, `factory`, and `remove` are reserved words that will not appear in a figure when quoted as a normal string. In order to display any of these words individually, type `'\reserved_word'` instead of `'reserved_word'`.

When the text `Interpreter` property is `tex` (the default), you can use a subset of TeX commands embedded in the string to produce special characters such as Greek letters and mathematical symbols. The following table lists these characters and the character sequences used to define them.

TeX Character Sequence Table

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\alpha	α	\upsilon	υ	\sim	~
\angle	∠	\phi	Φ	\leq	≤
\ast	*	\chi	χ	\infty	∞
\beta	β	\psi	ψ	\clubsuit	♣
\gamma	γ	\omega	ω	\diamondsuit	♦
\delta	δ	\Gamma	Γ	\heartsuit	♥
\epsilon	ε	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	Θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta		\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon		\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	∍	\partial	∂

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅
\cap	∩	\in		\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	o
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	´
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨			\copyright	©
\langle	〈				

You can also specify stream modifiers that control font type and color. The first four modifiers are mutually exclusive. However, you can use `\fontname` in combination with one of the other modifiers.

`Units`
`{normalized} | inches | centimeters | characters | points | pixels`

Position units. MATLAB uses this property to determine the units used by the `Position` property. All positions are measured from the lower left corner of the figure window.

- `normalized` — Interpret `Position` as a fraction of the width and height of the parent axes. When you resize the axes, MATLAB modifies the size of the object accordingly.
- `pixels`, `inches`, `centimeters`, and `points` — Absolute units. 1 point = $^1/_{72}$ inch.
- `characters` — Based on the size of characters in the default system font. The width of one `characters` unit is the width of the letter `x`, and the height of one `characters` unit is the distance between the baselines of two lines of text.

`VerticalAlignment`
`top | cap | {middle} | baseline | bottom`

Vertical alignment of text. Specifies the vertical justification of the text string within the textbox. It determines where MATLAB places the string vertically with regard to the points specified by the `Position` property.

Note that `top` and `cap` both place the text at the top of the box, while `baseline` and `bottom` both align the text on the bottom.

See Also

`annotation`

ans

Purpose

Most recent answer

Syntax

```
ans
```

Description

The MATLAB software creates the `ans` variable automatically when you specify no output argument.

Examples

The statement

```
2+2
```

is the same as

```
ans = 2+2
```

See Also

`display`

Purpose

Determine if any array elements are nonzero

Syntax

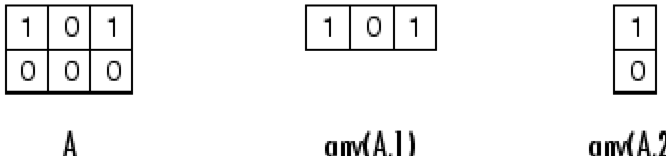
```
B = any(A)
B = any(A,dim)
```

Description

`B = any(A)` tests whether *any* of the elements along various dimensions of an array is a nonzero number or is logical `1` (`true`). The `any` function ignores entries that are `NaN` (Not a Number).

- If `A` is a vector, `any(A)` returns logical `1` (`true`) if any of the elements of `A` is a nonzero number or is logical `1` (`true`), and returns logical `0` (`false`) if all the elements are zero.
- If `A` is a nonempty matrix, `any(A)` treats the columns of `A` as vectors, returning a row vector of logical `1`'s and `0`'s.
- If `A` is an empty 0-by-0 matrix, `any(A)` returns logical `0` (`false`).
- If `A` is a multidimensional array, `any(A)` acts along the first nonsingleton dimension and returns an array of logical values. The size of this dimension reduces to `1` while the sizes of all other dimensions remain the same.

`B = any(A,dim)` tests along the dimension of `A` specified by scalar *`dim`*.



Examples

Example 1 – Reducing a Logical Vector to a Scalar Condition

Given

```
A = [0.53 0.67 0.01 0.38 0.07 0.42 0.69]
```

then `B = (A < 0.5)` returns logical `1` (`true`) only where `A` is less than one half:

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

The `any` function reduces such a vector of logical conditions to a single condition. In this case, `any(B)` yields logical `1`.

This makes `any` particularly useful in `if` statements:

```
if any(A < 0.5)do something
end
```

where code is executed depending on a single condition, not a vector of possibly conflicting conditions.

Example 2– Reducing a Logical Matrix to a Scalar Condition

To determine if any elements in an entire array are nonzero, use `any` in conjunction with the colon operator.

```
A = eye(3);
any(A(:))

ans =

     1
```

Example 3 – Testing Arrays of Any Dimension

You can use the following type of statement on an array of any dimensions. This example tests a 3-D array to see if any of its elements are greater than 3:

```
x = rand(3,7,5) * 5;

any(x(:) > 3)
ans =
     1
```

or less than zero:

```
any(x(:) < 0)
ans =
```

```
0
```

Definitions

First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to `1`.

For example:

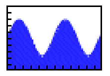
- If `X` is a 1-by-n row vector, then the second dimension is the first nonsingleton dimension of `X`.
- If `X` is a 1-by-0-by-n empty array, then the second dimension is the first nonsingleton dimension of `X`.
- If `X` is a 1-by-1-by-3 array, then the third dimension is the first nonsingleton dimension of `X`.

See Also

`all` | `colon (:)` | `max` | `mean` | `median` | `min` | `prod` | `std` | `sum` | `trapz` | Logical Operators: Elementwise | Logical Operators: Short-circuit | Relational Operators

Purpose

Filled area 2-D plot



Syntax

```
area(Y)
area(X,Y)
area(...,basevalue)
area(...,'PropertyName',PropertyValue,...)
area(axes_handle,...)
h = area(...)
```

Description

An area graph displays elements in `Y` as one or more curves and fills the area beneath each curve. When `Y` is a matrix, the curves are stacked showing the relative contribution of each row element to the total height of the curve at each x interval.

`area(Y)` plots the vector `Y` or the sum of each column in matrix `Y`. The *x*-axis automatically scales to `1:size(Y,1)`.

`area(X,Y)` For vectors `X` and `Y`, `area(X,Y)` is the same as `plot(X,Y)` except that the area between `0` and `Y` is filled. When `Y` is a matrix, `area(X,Y)` plots the columns of `Y` as filled areas. For each `X`, the net result is the sum of corresponding values from the columns of `Y`.

If `X` is a vector, `length(X)` must equal `length(Y)`. If `X` is a matrix, `size(X)` must equal `size(Y)`.

`area(...,basevalue)` specifies the base value for the area fill. The default `basevalue` is `0`. See the `BaseValue` property for more information.

`area(...,'`*`PropertyName`*`',PropertyValue,...)` specifies property name and property value pairs for the patch graphics object created by `area`.

`area(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = area(...)` returns handles of `areaseries` graphics objects.

Creating an area graph of an m-by-n matrix creates n areaseries objects (that is, one per column), whereas a 1-by-n vector creates one areaseries object.

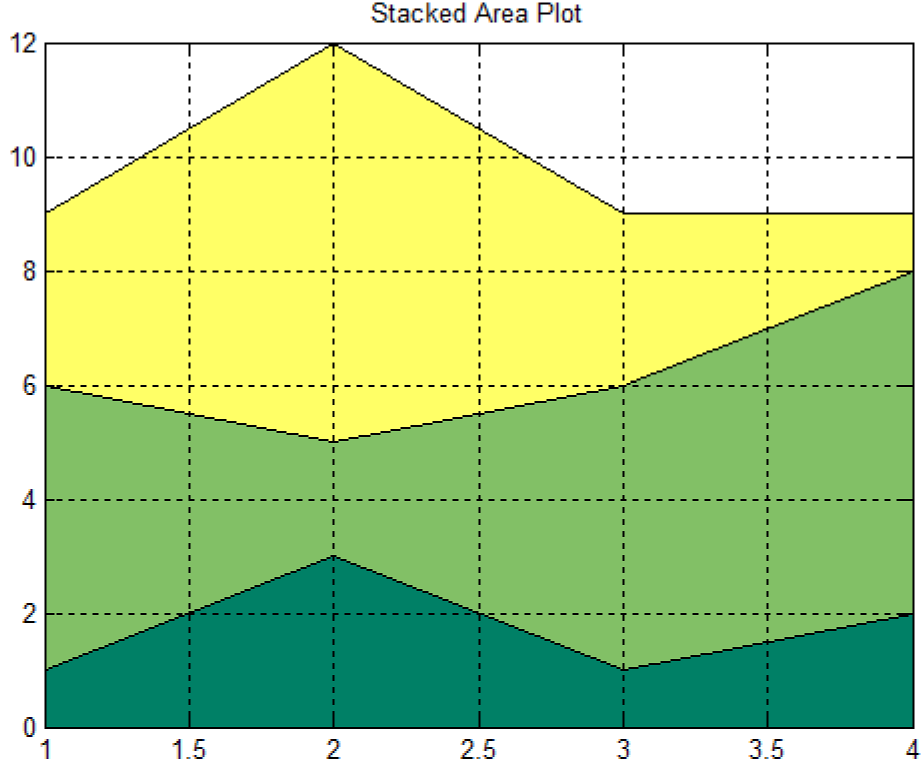
Some areaseries object properties that you set on an individual areaseries object set the values for all areaseries objects in the graph. See the property descriptions for information on specific properties.

Examples

Stacked Area Graph

This example plots the data in the variable `Y` as an area graph. Each subsequent column of `Y` is stacked on top of the previous data. The figure colormap controls the coloring of the individual areas. You can explicitly set the color of an area using the `EdgeColor` and `FaceColor` properties.

```
Y = [1, 5, 3;
 3, 2, 7;
 1, 5, 3;
 2, 6, 1];
area(Y)
grid on
colormap summer
set(gca,'Layer','top')
title 'Stacked Area Plot'
```

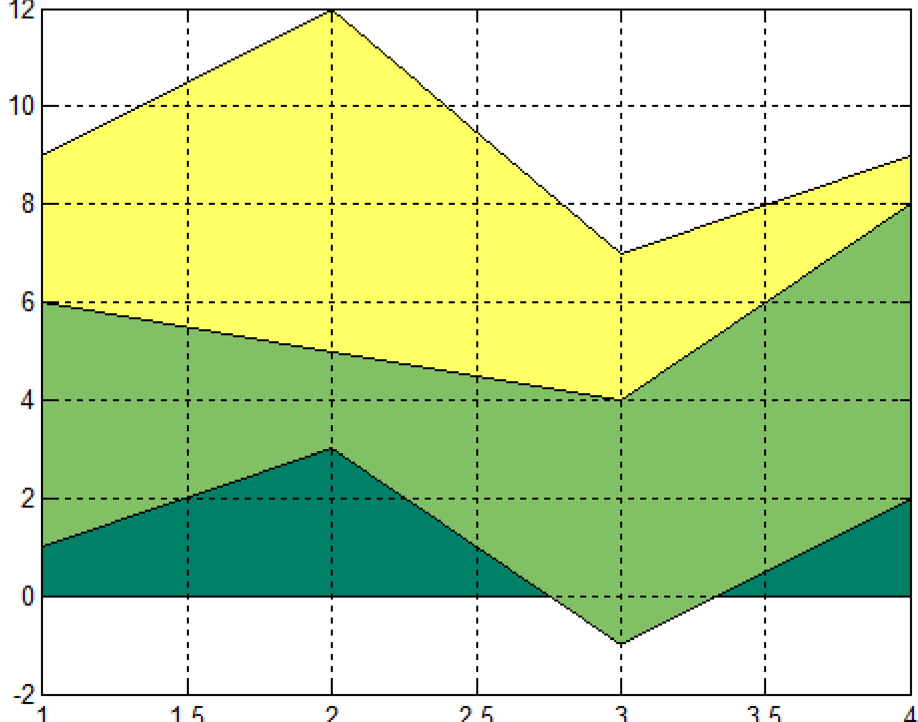


Adjusting the Base Value

The area function uses a *y*-axis value of 0 as the base of the filled areas. You can change this value by setting the area `BaseValue` property. For example, negate one of the values of Y from the previous example and replot the data.

```
Y(3,1) = -1; % Was 1
h = area(Y);
set(gca,'Layer','top')
grid on
colormap summer
```

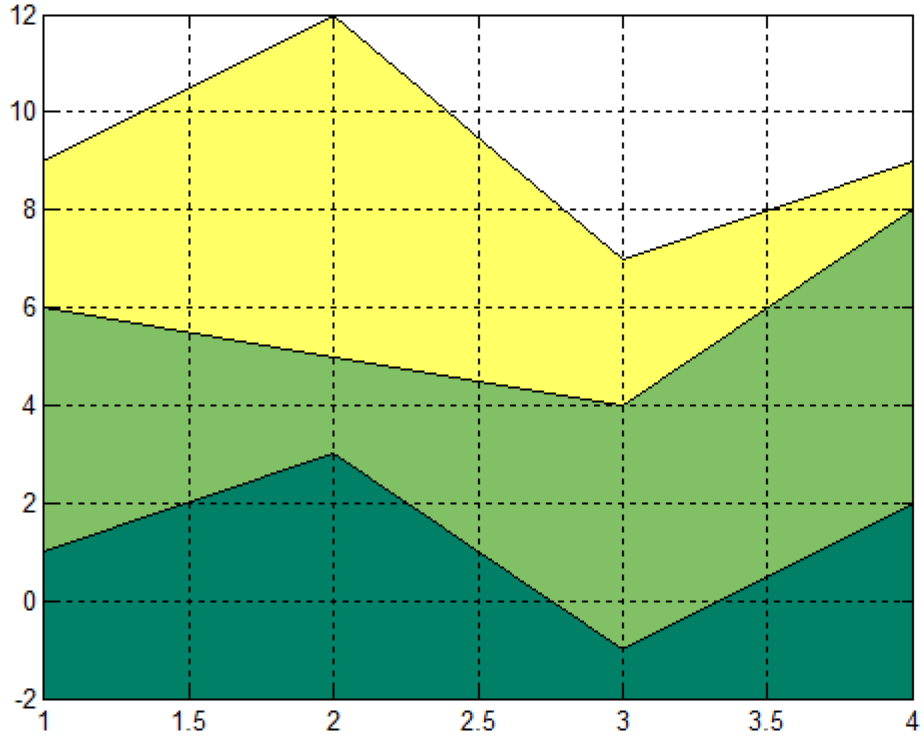
The area graph now looks like this:



Adjusting the `BaseValue` property improves the appearance of the graph:

```
set(h,'BaseValue',-2)
```

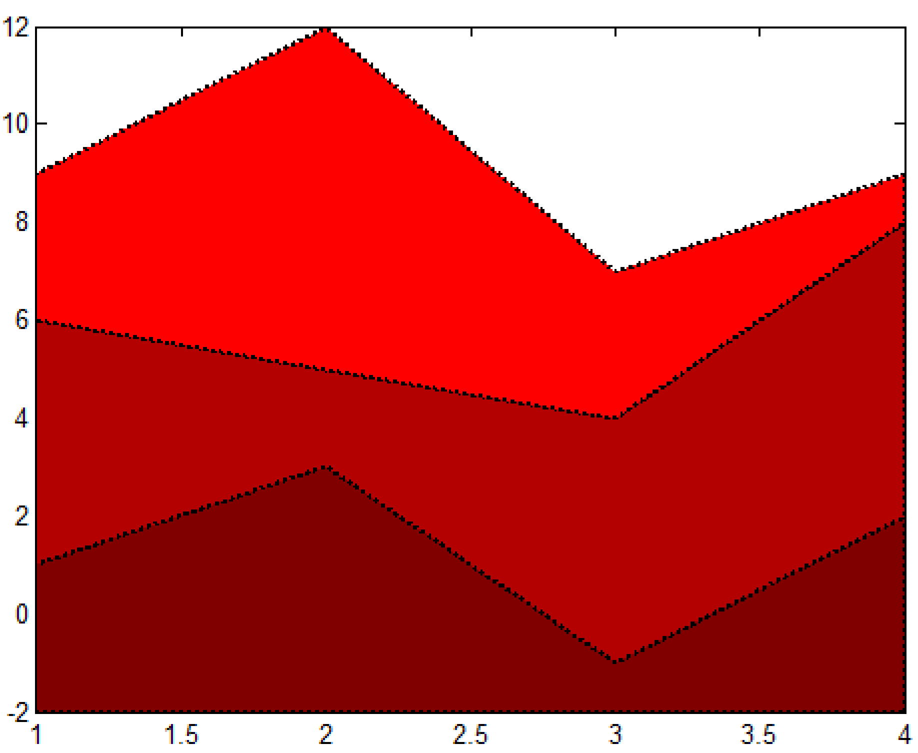
Setting the `BaseValue` property on one areaseries object sets the values of all objects.



Specifying Colors and Line Styles

You can specify the colors of the filled areas and the type of lines used to separate them.

```
h = area(Y,-2); % Set BaseValue via argument
set(h(1),'FaceColor',[.5 0 0])
set(h(2),'FaceColor',[.7 0 0])
set(h(3),'FaceColor',[1 0 0])
set(h,'LineStyle',':','LineWidth',2) % Set
all to same value
```



See Also

`bar` | `plot` | `sort` | "Pie Charts, Bar Plots, and Histograms" | `Areaseries Properties`

How To

- "Area Graphs"

Areaseries Properties

Purpose

Define areaseries properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands or with the property editor (`propertyeditor`).

Note that you cannot define default properties for areaseries objects.

See "Plot Objects" for more information on areaseries objects.

Areaseries Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`Annotation`

`hg.Annotation` object (read-only)

Control the display of areaseries objects in legends. Specifies whether this areaseries object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the areaseries object is displayed in a figure legend.

IconDisplayStyle Value	Purpose
`on`	Include the areaseries object in a legend as one entry, but not its children objects
`off`	Do not include the areaseries or its children in a legend (default)
`children`	Include only the children of the areaseries as separate entries in the legend

Setting the IconDisplayStyle Property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the IconDisplayStyle Property

See "Controlling Legends" for more information and examples.

`BaseValue`

double: y-axis value

Value where filled area base is drawn. Specify the value along the y-axis at which the MATLAB software draws the baseline of the bottommost filled area. The default is `0`.

`BeingDeleted`

`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`
string | function handle

Button press callback function. Executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be:

- A string that is a valid MATLAB expression
- The name of a MATLAB file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`Children`

array of graphics object handles

Children of this object. The handle of a patch object that is the child of this object (whether visible or not).

If a child object's `HandleVisibility` property is `callback` or `off`, its handle does not show up in this object's `Children` property. If you want the handle in the `Children` property, set the root `ShowHiddenHandles` property to `on`. For example:

```
set(0,'ShowHiddenHandles','on')
```

`Clipping`

`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs appear outside the axes plot box. This occurs if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`CreateFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback function executed during object creation. Defines a callback that executes when MATLAB creates an object. The default is an empty array.

You must specify the callback during the creation of the object. For example:

```
graphicfcn(y,'CreateFcn',@CallbackFcn)
```

where *@CallbackFcn* is a function handle that references the callback function and *graphicfcn* is the plotting function which creates this object.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DeleteFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback executed during object deletion. Executes when this object is deleted (for example, this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values. The default is an empty array.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`

string

String used by legend. The `legend` function uses the `DisplayName` property to label the areaseries object in the legend. The default is an empty string.

- If you specify string arguments with the `legend` function, MATLAB set `DisplayName` to the corresponding string and uses that string for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, MATLAB set `DisplayName` to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add a legend programmatically that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more information and examples.

`EdgeColor`

`{[0 0 0]} | none | ColorSpec`

Color of line separating filled areas. You can set the color of the edges of filled areas to a three-element RGB vector or one of the MATLAB predefined names, including the string `none`. The default value is `[0 0 0]` (`black`). See the `ColorSpec` reference page for more information on specifying color.

`EraseMode`

`{normal} | none | xor | background`

Erase mode. Controls the technique MATLAB uses to draw and erase objects. Use alternative erase modes for creating animated

sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the object when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it because MATLAB stores no information about its former location.
- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object's color depends on the color of whatever is beneath it on the display.
- `background` — Erase the object by redrawing it in the axes background `Color`, or the figure background `Color` if the axes `Color` property is `none`. This damages objects that are behind the erased object, but properly colors the erased object.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors (for example, performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`FaceColor`
`{flat} | none | ColorSpec`

Color of filled areas.

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for all filled areas.
- `none` — Do not draw faces. Note that MATLAB draws `EdgeColor` independently of `FaceColor`.
- `flat` — The object uses the figure `colormap` to determine the color of the filled areas.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible.
- `callback` — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties. See also `findall`.

Handle Validity

Hidden handles are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`

`{on} | off`

Selectable by mouse click. Determines whether this object can become the current object (as returned by the gco command and the figure CurrentObject property) as a result of a mouse click on the objects that compose the area graph. If HitTest is off, clicking this object selects the object below it (which is usually the axes containing it).

HitTestArea
on | {off}

Select areaseries object on filled area or extent of graph. Select plot objects by:

- Clicking an area (default).
- Clicking anywhere in the extent of the area plot.

When HitTestArea is off, you must click an area to select the areaseries object. When HitTestArea is on, you can select the areaseries object by clicking anywhere within the extent of the area graph (that is, anywhere within a rectangle that encloses all the area plots).

Interruptible
off | {on}

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the Interruptible property affects only the callbacks for theButtonDownFcn property. A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both the callbacks based on the Interruptible property of the object of the *running* callback.

When the Interruptible property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see "Control Callback Execution and Interruption".

`LineStyle`

`{-} | -- | : | -. | none`

Line style of edges of filled areas.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line
'none'	No line

`LineWidth`

size in points

Width of linear objects and edges of filled areas. Specify in points. 1 point = $^1/_{72}$ inch. The default is 0.5 points.

`Parent`

handle of parent axes, hggroup, or hgtransform

Parent of object. Handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`

on | {off}

Object selection state. When you set this property to on, MATLAB displays selection handles at the corners and midpoints if the SelectionHighlight property is also on (the default). You can, for example, define the ButtonDownFcn callback to set this property to on, thereby indicating that this particular object is selected. This property is also set to on when an object is manually selected in plot edit mode.

`SelectionHighlight`
`{on} | off`

Object highlighted when selected.

- `on` — MATLAB indicates the selected state by drawing four edge handles and four corner handles.
- `off` — MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`Tag`
string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

For example, create an areaseries object and set the `Tag` property.

```
t = area(Y,'Tag','area1')
```

When you want to access objects of a given type, use `findobj` to find the object's handle. The following statement changes the `FaceColor` property of the object whose `Tag` is `area1`.

```
set(findobj('Tag','area1'),'FaceColor','red')
```

`Type`
string (read-only)

Type of graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For areaseries objects, `Type` is '`hggroup`'.

The following statement finds all the hggroup objects in the current axes.

```
t = findobj(gca,'Type','hggroup');
```

`UIContextMenu`
handle of uicontextmenu object

Associate context menu with object. Handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object. The default value is an empty array.

`UserData`
array

User-specified data. Data you want to associate with this object (including cell arrays and structures). The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`
`{on} | off`

Visibility of object and its children.

- `on` — Object and all children of the object are visible unless the child object's `Visible` property is `off`.
- `off` — Object not displayed. However, the object still exists and you can set and query its properties.

`XData`
vector | matrix

x-axis values for graph. The *x*-axis values for graphs are specified by the `X` input argument. If `XData` is a vector, `length(XData)` must equal `length(YData)` and must be monotonic. If `XData` is a

matrix, `size(XData)` must equal `size(YData)` and each column must be monotonic.

You can use `XData` to define meaningful coordinates for an underlying surface whose topography is being mapped. See “Changing the Offset of a Contour” for more information.

`XDataMode`

`{auto} | manual`

Use automatic or user-specified x-axis values. If you specify `XData` (by setting the `XData` property or specifying the `X` input argument), MATLAB sets this property to `manual` and uses the specified values to label the *x*-axis.

If you set `XDataMode` to `auto` after specifying `XData`, MATLAB resets the *x*-axis ticks to `1:size(YData,1)` or to the column indices of the `ZData`, overwriting any previous values for `XData`.

`XDataSource`

MATLAB variable, as a string

Link `XData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `XData`. The default value is an empty array.

```
set(h,'XDataSource','xdatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object’s `XDataSource` does not change the object’s `XData` values, but you can use `refreshdata` to force an update of the object’s data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`YData`

vector | matrix

Area plot data. `YData` contains the data plotted as filled areas (the `Y` input argument). If `YData` is a vector, `area` creates a single filled area whose upper boundary is defined by the elements of `YData`. If `YData` is a matrix, `area` creates one filled area per column, stacking each on the previous plot. The default value is the `Y` input argument.

The input argument `Y` in the `area` function calling syntax assigns values to `YData`.

`YDataSource`

MATLAB variable, as a string

Link `YData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `YData`. The default value is an empty array.

```
set(h,'YDataSource','Ydatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `YDataSource` does not change the object's `YData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

How To

- "Bar and Area Graphs"
- "Plot Objects"
- "Setting Default Property Values"

Purpose

Apply function to each element of array

Syntax

```
[B1,...,Bm] = arrayfun(func,A1,...,An)
[B1,...,Bm] = arrayfun(func,A1,...,An,Name,Value)
```

Description

`[B1,...,Bm] = arrayfun(func,A1,...,An)` calls the function specified by function handle `func` and passes elements from arrays `A1,...,An`, where `n` is the number of inputs to function `func`. Output arrays `B1,...,Bm`, where `m` is the number of outputs from function `func`, contain the combined outputs from the function calls. The `i`th iteration corresponds to the syntax `[B1(i),...,Bm(i)] = func(A1{i},...,An{i})`. The `arrayfun` function does not perform the calls to function `func` in a specific order.

`[B1,...,Bm] = arrayfun(func,A1,...,An,Name,Value)` calls function `func` with additional options specified by one or more `Name,Value` pair arguments. Possible values for `Name` are `'UniformOutput'` or `'ErrorHandler'`.

Input Arguments

func

Handle to a function that accepts `n` input arguments and returns `m` output arguments.

If function `func` corresponds to more than one function file (that is, if `func` represents a set of overloaded functions), MATLAB determines which function to call based on the class of the input arguments.

A1,...,An

Arrays that contain the `n` inputs required for function `func`. Each array must have the same dimensions. Arrays can be numeric, character, logical, cell, structure, or user-defined object arrays.

If any input `A` is a user-defined object array, and you overloaded the `subsref` or `size` methods, `arrayfun` requires that:

- The `size` method returns an array of type `double`.
- The object array supports linear indexing.

- The product of the sizes returned by the `size` method does not exceed the limit of the array, as defined by linear indexing into the array.

Name-Value Pair Arguments

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

'UniformOutput'

Logical value, as follows:

`true (1)`	Indicates that for all inputs, each output from function `func` is a cell array or a scalar value that is always of the same type. The `arrayfun` function combines the outputs in arrays `B1,...,Bm`, where `m` is the number of function outputs. Each output array is of the same type as the individual function outputs.
`false (0)`	Requests that the `arrayfun` function combine the outputs into cell arrays `B1,...,Bm`. The outputs of function `func` can be of any size or type.

Default: `true`

'ErrorHandler'

Handle to a function that catches any errors that occur when MATLAB attempts to execute function `func`. Define this function so that it rethrows the error or returns valid outputs for function `func`.

MATLAB calls the specified error-handling function with two input arguments:

- A structure with these fields:

`identifier`	Error identifier.
`message`	Error message text.
`index`	Linear index corresponding to the element of the input cell array at the time of the error.

- The set of input arguments to function `func` at the time of the error.

Output Arguments

B1,...,Bm

Arrays that collect the `m` outputs from function `func`. Each array `B` is the same size as each of the inputs `A1,...,An`.

Function `func` can return output arguments of different classes. However, if `UniformOutput` is `true` (the default):

- The individual outputs from function `func` must be scalar values (numeric, logical, character, or structure) or cell arrays.
- The class of a particular output argument must be the same for each set of inputs. The class of the corresponding output array is the same as the class of the outputs from function `func`.

Examples

To run the examples in this section, create a nonscalar structure array with arrays of different sizes in field `f1`.

```
s(1).f1 = rand(3, 6);
s(2).f1 = magic(12);
s(3).f1 = ones(5, 10);
```

Count the number of elements in each `f1` field.

```
counts = arrayfun(@(x) numel(x.f1), s)
```

The syntax `@(x)` creates an anonymous function. This code returns

```
counts =
```

```
    18   144    50
```

Compute the size of each array in the `f1` fields.

```
[nrows, ncols] = arrayfun(@(x) size(x.f1), s)
```

This code returns

```
nrows =
     3    12     5
ncols =
     6    12    10
```

Compute the mean of each column in the `f1` fields of `s`. Because the output is nonscalar, set `UniformOutput` to `false`.

```
averages = arrayfun(@(x) mean(x.f1), s, 'UniformOutput', false)
```

This code returns

```
averages =
    [1x6 double]    [1x12 double]    [1x10 double]
```

Create additional nonscalar structures `t` and `u`, and test for equality between the arrays in fields `f1` across structures `s`, `t`, and `u`.

```
t = s;  t(1).f1(:)=0;
u = s;  u(2).f1(:)=0;

same = arrayfun(@(x,y,z) isequal(x.f1, y.f1, z.f1), s, t, u)
```

This code returns

```
same =
     0     0     1
```

See Also

`structfun` | `cellfun` | `spfun` | `function_handle` | `cell2mat`

Tutorials

- "Anonymous Functions"

FTP.ascii

Purpose

Set FTP transfer type to ASCII

Syntax

```
ascii(ftpobj)
```

Description

`ascii(ftpobj)` sets the download and upload FTP mode to ASCII, which converts new line characters. Use this method only for text files, including HTML pages and Rich Text Format (RTF) files.

Input Arguments

ftpobj

FTP object created by `ftp`.

Examples

Connect to the MathWorks FTP server, and switch from binary (default) to ASCII mode:

```
mw=ftp('ftp.mathworks.com');
ascii(mw)
```

See Also

`binary` | `ftp`

Purpose

Inverse secant in radians

Syntax

```
Y = asec(X)
```

Description

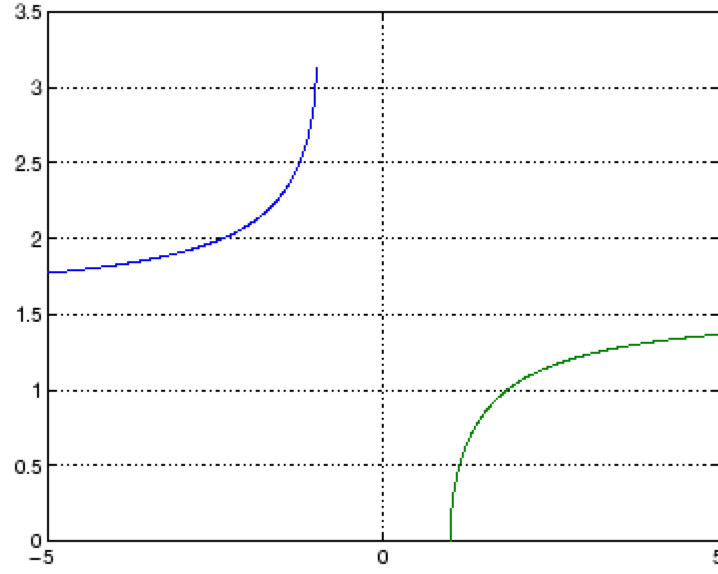
`Y = asec(X)` returns the inverse secant (arcsecant) for each element of `X`.

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

Examples

Graph the inverse secant over the domains $1 \le x \le 5$ and $-5 \le x \le -1$.

```
x1 = -5:0.01:-1;
x2 = 1:0.01:5;
plot(x1,asec(x1),x2,asec(x2)), grid on
```



See Also

`asecd` | `asech` | `sec`

asecd

Purpose

Inverse secant in degrees

Syntax

```
Y = asecd(X)
```

Description

`Y = asecd(X)` returns the inverse secant of the elements of `X` in degrees.

Input Arguments

X - Secant of angle

scalar value | vector | matrix | N-D array

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

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

Output Arguments

Y - Angle in degrees

scalar value | vector | matrix | N-D array

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

Examples

Inverse Secant of Vector

```
x = [10 1 Inf];
y = asecd(x)
```

```
y =

   84.2608         0   90.0000
```

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

Inverse Secant of Complex Value

```
asecd(1+i)
```

```
ans =

  64.0864 +30.4033i
```

See Also

secd | asec | sec

asech

Purpose

Inverse hyperbolic secant

Syntax

```
Y = asech(X)
```

Description

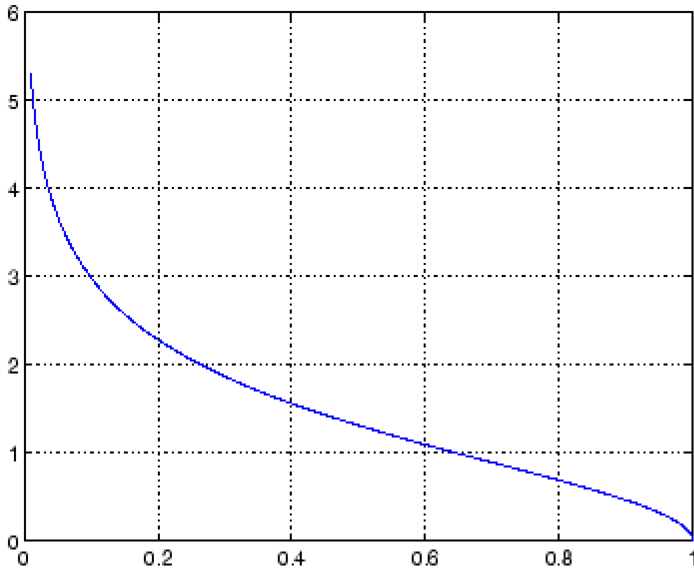
`Y = asech(X)` returns the inverse hyperbolic secant for each element of `X`.

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

Examples

Graph the inverse hyperbolic secant over the domain $0.01 \le x \le 1$.

```
x = 0.01:0.001:1;
plot(x,asech(x)), grid on
```



See Also

`asec` | `sech` | `asinh` | `acosh`

Purpose Inverse sine in radians

Syntax

```
Y = asin(X)
```

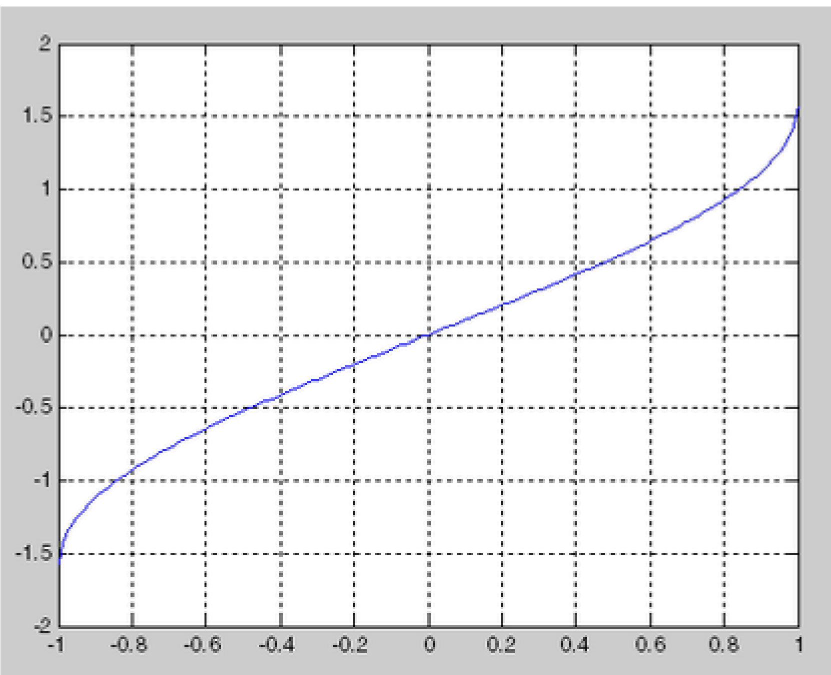
Description `Y = asin(X)` returns the inverse sine (arcsine) for each element of `X`. The `asin` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians. For real elements of `X` in the domain `[-1,1]`, `asin(X)` is in the range

$$\left[-\frac{\pi}{2},\frac{\pi}{2}\right].$$

For real elements of `x` outside the range `[-1,1]`, `asin(X)` is complex.

Examples Graph the inverse sine function over the domain $-1 \leq x \leq 1$.

```
x = -1:.01:1;
plot(x,asin(x)), grid on
```



See Also

`asind` | `sin` | `sind`

asind

Purpose

Inverse sine in degrees

Syntax

```
Y = asind(X)
```

Description

`Y = asind(X)` returns the inverse sine of the elements of `X` in degrees.

Input Arguments

X - Sine of angle
scalar value | vector | matrix | N-D array

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

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

Output Arguments

Y - Angle in degrees
scalar value | vector | matrix | N-D array

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

Examples

Inverse sine of one

Show that the inverse sine of 1 is exactly 90°.

```
asind(1)
```

```
ans =

    90
```

Round Trip Calculation for Complex Angles

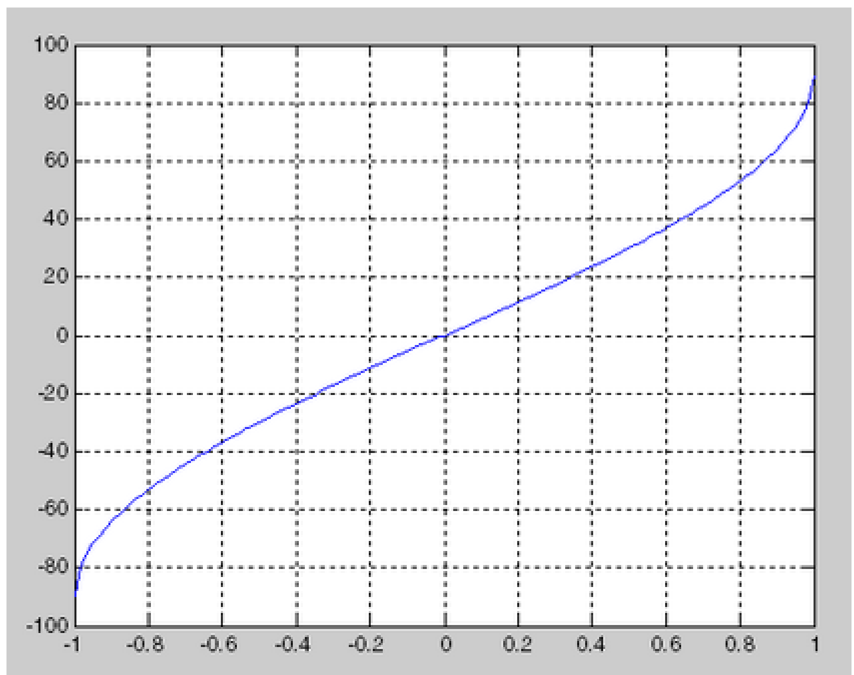
Show that inverse sine, followed by sine, returns the original values of `X`.

```
sind(asind([2 3]))
```

```
ans =

    2.0000    3.0000
```

Graph inverse sine over the interval [-1,1]

```
x = -1:.01:1;
plot(x,asind(x))
grid on
```



See Also

`sind` | `sin` | `asin`

Purpose

Inverse hyperbolic sine

Syntax

```
Y = asinh(X)
```

Description

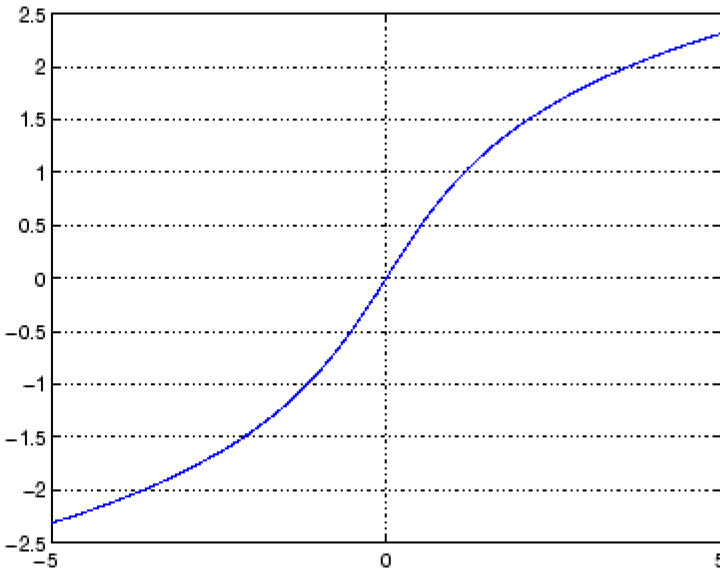
`Y = asinh(X)` returns the inverse hyperbolic sine for each element of `X`.

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

Examples

Graph the inverse hyperbolic sine function over the domain $-5 \leq x \leq 5$.

```
x = -5:.01:5;
plot(x,asinh(x)), grid on
```



See Also

`asin` | `sinh` | `acosh`

assert

Purpose

Generate error when condition is violated

Syntax

```
assert(expression)
assert(expression, 'msgString')
assert(expression, 'msgString', value1, value2, ...)
assert(expression, 'msgIdent', 'msgString', value1, value2,
   ...)
```

Description

`assert(expression)` evaluates *expression* and, if it is `false`, generates an exception.

`assert(expression, 'msgString')` evaluates *expression* and, if it is `false`, generates an exception and displays the string contained in *msgString*. This string must be enclosed in single quotation marks. When *msgString* is the last input to `assert`, the MATLAB software displays it literally, without performing any substitutions on the characters in *msgString*.

`assert(expression, 'msgString', value1, value2, ...)` evaluates *expression* and, if it is `false`, generates an exception and displays the formatted string contained in *msgString*. The *msgString* string can include escape sequences such as `\t` or `\n`, as well as any of the C language conversion operators supported by the `sprintf` function (e.g., `%s` or `%d`). Additional arguments *value1*, *value2*, etc. provide values that correspond to and replace the conversion operators.

See "Formatting Strings" in the MATLAB Programming Fundamentals documentation for more detailed information on using string formatting commands.

MATLAB makes substitutions for escape sequences and conversion operators in *msgString* in the same way that it does for the `sprintf` function.

`assert(expression, 'msgIdent', 'msgString', value1, value2, ...)` evaluates *expression* and, if it is `false`, generates an exception and displays the formatted string *msgString*, also tagging the error with the message identifier *msgIdent*. See "Message Identifiers" in the MATLAB Programming Fundamentals documentation for information.

Examples

This function tests input arguments using assert:

```
function write2file(varargin)
min_inputs = 3;
assert(nargin >= min_inputs, ...
  'You must call function %s with at least %d inputs', ...
  mfilename, min_inputs)

infile = varargin{1};
assert(ischar(infile), ...
      'First argument must be a filename.')
assert(exist(infile)~=0, 'File %s not found.', infile)

fid = fopen(infile, 'w');
assert(fid > 0, 'Cannot open file %s for writing', infile)

fwrite(fid, varargin{2}, varargin{3});
```

See Also

error | eval | try | dbstop | errordlg | warning | warndlg | MException | throw(MException) | rethrow(MException) | throwAsCaller(MException) | addCause(MException) | getReport(MException) | last(MException)

assignin

Purpose

Assign value to variable in specified workspace

Syntax

```
assignin(ws, 'var', val)
```

Description

`assignin(ws, 'var', val)` assigns the value `val` to the variable *var* in the workspace `ws`. The *var* input must be the array name only; it cannot contain array indices. If *var* does not exist in the specified workspace, `assignin` creates it. `ws` can have a value of `'base'` or `'caller'` to denote the MATLAB base workspace or the workspace of the caller function.

The `assignin` function is particularly useful for these tasks:

- Exporting data from a function to the MATLAB workspace
- Within a function, changing the value of a variable that is defined in the workspace of the caller function (such as a variable in the function argument list)

Tips

The MATLAB base workspace is the workspace that is seen from the MATLAB command line (when not in the debugger). The caller workspace is the workspace of the function that called the currently running function. Note that the base and caller workspaces are equivalent in the context of a function that is invoked from the MATLAB command line. For more information, see “Base and Function Workspaces”.

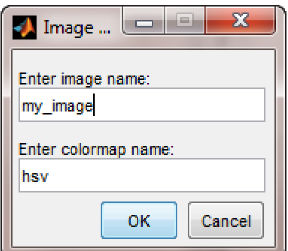
Examples

Example 1

This example creates a dialog box for the image display function, prompting a user for an image name and a colormap name. The `assignin` function is used to export the user-entered values to the MATLAB workspace variables `imfile` and `cmap`.

```
prompt = {'Enter image name:', 'Enter colormap name:'};
title = 'Image display - assignin example';
lines = 1;
def = {'my_image', 'hsv'};
answer = inputdlg(prompt, title, lines, def);
```

```
assignin('base', 'imfile', answer{1});
assignin('base', 'cmap', answer{2});
```



Example 2

`assignin` does not assign to specific elements of an array. The following statement generates an error:

```
X = 1:8;
assignin('base', 'X(3:5)', -1);
```

However, you can use the `evalin` function to do this:

```
evalin('base','X(3:5) = -1')
X =
     1     2    -1    -1    -1     6     7     8
```

See Also

`evalin`

Purpose Inverse tangent in radians

Syntax

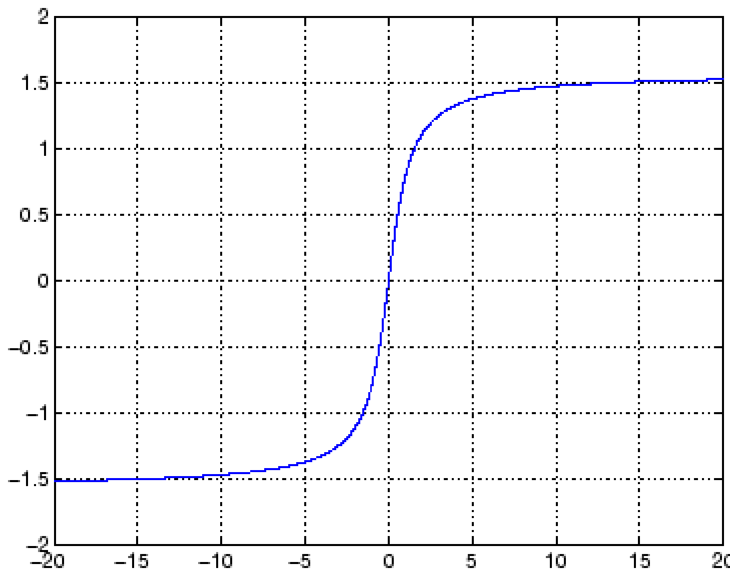
```
Y = atan(X)
```

Description `Y = atan(X)` returns the inverse tangent (arctangent) for each element of `X`. For real elements of `X`, `atan(X)` is in the range $[-\pi/2, \pi/2]$.

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

Examples Graph the inverse tangent function over the domain $-20 \le x \le 20$.

```
x = -20:0.01:20;
plot(x,atan(x)), grid on
```



See Also `atan2` | `tan` | `atand` | `atanh`

Purpose

Four-quadrant inverse tangent

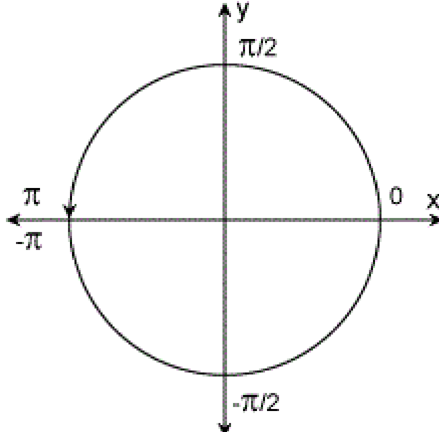
Syntax

```
P = atan2(Y,X)
```

Description

`P = atan2(Y,X)` returns an array `P` the same size as `X` and `Y` containing the element-by-element, four-quadrant inverse tangent (arctangent) of `Y` and `X`, which must be real.

Elements of `P` lie in the closed interval `[-pi,pi]`, where `pi` is the MATLAB floating-point representation of π. `atan` uses `sign(Y)` and `sign(X)` to determine the specific quadrant.



`atan2(Y,X)` contrasts with `atan(Y/X)`, whose results are limited to the interval $[-\pi/2, \pi/2]$, or the right side of this diagram.

Examples

Any complex number $z = x + iy$ is converted to polar coordinates with

```
r = abs(z)
theta = atan2(imag(z),real(z))
```

For example,

```
z = 4 + 3i;
```

```
r = abs(z)
theta = atan2(imag(z),real(z))

r =
     5

theta =
    0.6435
```

This is a common operation, so MATLAB software provides a function, `angle(z)`, that computes `theta = atan2(imag(z),real(z))`.

To convert back to the original complex number

```
z = r * exp(i * theta)
z =

   4.0000 + 3.0000i
```

See Also

`angle` | `atan` | `atanh` | `atan2d`

Purpose

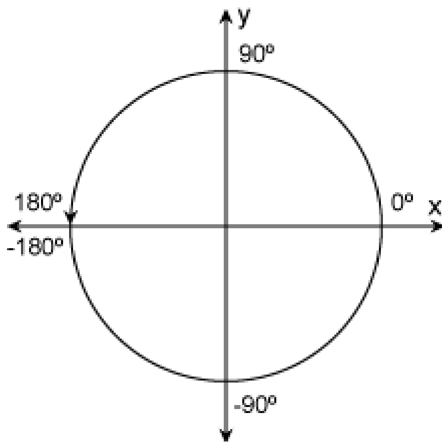
Four-quadrant inverse tangent in degrees

Syntax

```
D = atan2d(Y,X)
```

Description

`D = atan2d(Y,X)` returns the four-quadrant inverse tangent of points specified in the *x*-*y* plane. The result, `D`, is expressed in degrees.



Input Arguments

Y - *y*-coordinates

scalar value | vector | matrix | N-D array

y-coordinates, specified as a real-valued scalar, vector, matrix, or N-D array.

Data Types

`single` | `double`

X - *x*-coordinates

scalar value | vector | matrix | N-D array

x-coordinates, specified as a real-valued scalar, vector, matrix, or N-D array.

Data Types
`single` | `double`

Output Arguments

D - Angles in degrees
scalar value | vector | matrix | N-D array

Angles in degrees, returned as a scalar, vector, matrix, or N-D array. These angles correspond to the points defined by `X` and `Y`, and they lie in the closed interval [–180,180].

Examples

Inverse Tangent of Four Points on the Unit Circle

```
x = [1 0 -1 0];
y = [0 1 0 -1];
d = atan2d(y,x)

d =

     0    90   180   -90
```

Tips

- Use `atand(Y/X)` for the inverse tangent with results on the interval [–90, 90].

See Also

`atan2` | `atan` | `atand` | `tan` | `tand`

Purpose

Inverse tangent in degrees

Syntax

```
Y = atand(X)
```

Description

`Y = atand(X)` returns the inverse tangent of the elements of `X` in degrees.

Input Arguments

X - Tangent of angle

scalar value | vector | matrix | N-D array

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

Data Types

`single` | `double`

Complex Number Support: Yes

Output Arguments

Y - Angle in degrees

scalar value | vector | matrix | N-D array

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

Examples

Inverse Tangent of Vector

```
x = [-50 -20 0 20 50];
y = atand(x)
```

```
y =

  -88.8542   -87.1376         0    87.1376    88.8542
```

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

Inverse Tangent of Complex Value

```
atand(10+i)
```

```
ans =

  84.3450 + 0.5618i
```

See Also

tand | atan | tan | atan2d

Purpose Inverse hyperbolic tangent

Syntax

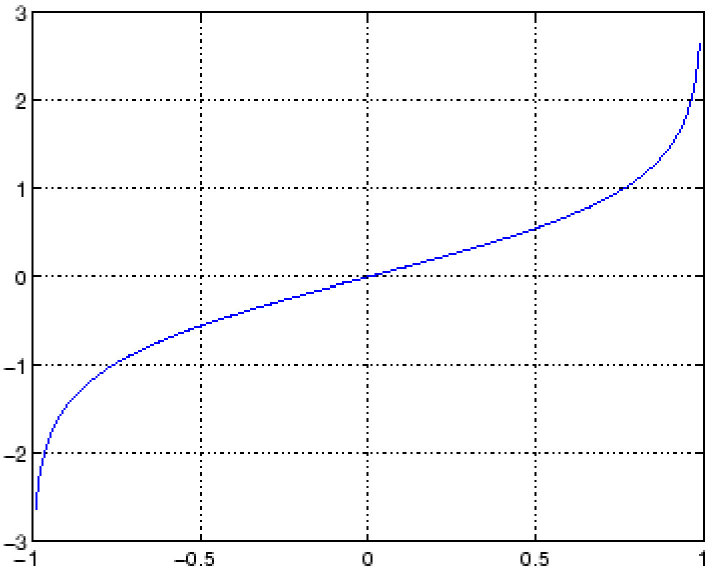
```
Y = atanh(X)
```

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

`Y = atanh(X)` returns the inverse hyperbolic tangent for each element of `X`.

Examples Graph the inverse hyperbolic tangent function over the domain $-1 < x < 1$.

```
x = -0.99:0.01:0.99;
plot(x,atanh(x)), grid on
```



See Also `atan2` | `atan` | `tanh` | `asinh` | `acosh` | `tan`

Purpose

Information about audio device

Syntax

```
devinfo = audiodevinfo
devs = audiodevinfo(IO)
name = audiodevinfo(IO, ID)
ID = audiodevinfo(IO, name)
DriverVersion = audiodevinfo(IO, ID, 'DriverVersion')
ID = audiodevinfo(IO, rate, bits, chans)
doesSupport = audiodevinfo(IO, ID, rate, bits, chans)
```

Description

`devinfo = audiodevinfo` returns a structure, `devinfo`, containing two fields, `input` and `output`. Each field is an array of structures, with each structure containing information about one of the audio input or output devices on the system. The individual device structure fields are:

- `Name` — A string indicating the name of the device.
- `DriverVersion` — A string indicating the type of the installed device driver.
- `ID` — The ID of the device.

`devs = audiodevinfo(IO)` returns the number of input or output audio devices on the system. Use an `IO` value of 1 to indicate input, and an `IO` value of 0 to indicate output.

`name = audiodevinfo(IO, ID)` returns the name of the input or output audio device identified by device `ID`.

`ID = audiodevinfo(IO, name)` returns the device `ID` of the input or output audio device identified by the given `name` (partial matching, case sensitive). If no audio device is found with the given name, -1 is returned.

`DriverVersion = audiodevinfo(IO, ID, 'DriverVersion')` returns a string indicating the type of the specified audio input or output device.

`ID = audiodevinfo(IO, rate, bits, chans)` returns the device `ID` of the first input or output device that supports the sample rate, number of bits, and number of channels specified by the values of `rate`, `bits`, and `chans`, respectively. If no supporting device is found, -1 is returned.

`doesSupport = audiodevinfo(IO, ID, rate, bits, chans)` returns 1 or 0 for whether or not the input or output audio device specified by `ID` can support the given sample rate, number of bits, and number of channels.

See Also

`audioplayer` | `audiorecorder`

audioplayer

Purpose

Create object for playing audio

Syntax

```
player = audioplayer(Y,Fs)
player = audioplayer(Y,Fs,nBits)
player = audioplayer(Y,Fs,nBits,ID)
player = audioplayer(recorder)
player = audioplayer(recorder,ID)
```

Description

`player = audioplayer(Y,Fs)` creates an `audioplayer` object for signal `Y`, using sample rate `Fs`. The function returns a handle to the `audioplayer` object, `player`.

`player = audioplayer(Y,Fs,nBits)` uses `nBits` bits per sample for signal `Y`.

`player = audioplayer(Y,Fs,nBits,ID)` uses the audio device identified by `ID` for output.

`player = audioplayer(recorder)` creates an `audioplayer` object using audio recorder object `recorder`.

`player = audioplayer(recorder,ID)` creates an object from `recorder` that uses the audio device identified by`ID` for output.

Input Arguments

Y

Audio signal represented by a vector or two-dimensional array containing `single`, `double`, `int8`, `uint8`, or `int16` values.

The value range of the input sample depends on the data type. The following table lists these ranges.

Data Type	Sample Value Range
`int8`	-128 to 127
`uint8`	0 to 255
`int16`	-32768 to 32767

Data Type	Sample Value Range
`single`	-1 to 1
`double`	-1 to 1

Fs

Sampling rate in Hz. Valid values depend on the specific audio hardware installed. Typical values supported by most sound cards are 8000, 11025, 22050, 44100, 48000, and 96000 Hz.

nBits

Bits per sample. Specify only when signal `Y` is represented by floating-point values. Valid values depend on the audio hardware installed: `8`, `16`, or `24`.

Default: `16`

ID

Device identifier. To obtain the ID of a device, use the `audiodevinfo` function.

Default: `-1` (default device)

recorder

Audio recorder object created by `audiorecorder`.

Methods

Note When calling any method, include the `audioplayer` object name using function syntax, such as `stop(`*`player`*`)`.

`get`	Query properties of `audioplayer` object.
`isplaying`	Query whether playback is in progress: returns `true` or `false`.

`pause`	Pause playback.
`play`	Play audio from beginning to end.
`playblocking`	Play, and do not return control until playback completes.
`resume`	Restart playback from paused position.
`set`	Set properties of `audioplayer` object.
`stop`	Stop playback.

See the reference pages for `get`, `play`, `playblocking`, and `set` for additional syntax options.

Properties

`BitsPerSample`	Number of bits per sample. (Read-only)
`CurrentSample`	Current sample that the audio output device is playing. If the device is not playing, `CurrentSample` is the next sample to play with `play` or `resume`. (Read-only)
`DeviceID`	Identifier for audio device. (Read-only)
`NumberOfChannels`	Number of audio channels. (Read-only)
`Running`	Status of the audio player: `'on'` or `'off'`. (Read-only)
`SampleRate`	Sampling frequency in Hz.
`TotalSamples`	Total length of the audio data in samples. (Read-only)
`Tag`	String that labels the object.
`Type`	Name of the class: `'audioplayer'`. (Read-only)
`UserData`	Any type of additional data to store with the object.

The following four properties apply to callback functions. The first two inputs to your callback function must be the `audioplayer` object and an *`event`* structure.

`StartFcn`	Function to execute one time when playback starts.
`StopFcn`	Function to execute one time when playback stops.
`TimerFcn`	Function to execute repeatedly during playback. To specify time intervals for the repetitions, use the `TimerPeriod` property.
`TimerPeriod`	Time in seconds between `TimerFcn` callbacks. Default: `.05`

Examples

Load and play a sample audio file of Handel's "Hallelujah Chorus:"

```
load handel;
player = audioplayer(y, Fs);
play(player);
```

See Also

`audiodevinfo` | `audiorecorder` | `sound`

How To

- "Characteristics of Audio Files"
- "Play Audio"

Purpose

Create object for recording audio

Syntax

```
recorder = audiorecorder
recorder = audiorecorder(Fs,nBits,nChannels)
recorder = audiorecorder(Fs,nBits,nChannels,ID)
```

Description

`recorder = audiorecorder` creates an 8000 Hz, 8-bit, 1-channel `audiorecorder` object.

`recorder = audiorecorder(Fs,nBits,nChannels)` sets the sample rate `Fs` (in Hz), the sample size `nBits`, and the number of channels `nChannels`.

`recorder = audiorecorder(Fs,nBits,nChannels,ID)` sets the audio input device to the device specified by `ID`.

Tips

- To use an `audiorecorder` object, your system must have a properly installed and configured sound card.
- `audiorecorder` is not intended for long, high-sample-rate recording. `audiorecorder` uses system memory for storage and does not use disk buffering. When you attempt a large recording, your MATLAB performance sometimes degrades over time.

Input Arguments

Fs

Sampling rate in Hz. Valid values depend on the specific audio hardware installed. Typical values supported by most sound cards are 8000, 11025, 22050, 44100, 48000, and 96000 Hz.

Default: 8000

nBits

Bits per sample. Valid values depend on the audio hardware installed: `8`, `16`, or `24`.

Default: 8

nChannels

The number of channels: `1` (mono) or `2` (stereo).

Default: 1

ID

Device identifier. To obtain the ID of a device, use the `audiodevinfo` function.

Default: `-1` (default device)

Methods

Note When calling any method, include the `audiorecorder` object name using function syntax, such as `stop(`*`recorder`*`)`.

`get`	Query properties of `audiorecorder` object.
`getaudiodata`	Create an array that stores the recorded signal values.
`getplayer`	Create an `audioplayer` object.
`isrecording`	Query whether recording is in progress: returns `true` or `false`.
`pause`	Pause recording.
`play`	Play recorded audio. This method returns an `audioplayer` object.
`record`	Start recording.
`recordblocking`	Record, and do not return control until recording completes. This method requires a second input for the length of the recording in seconds: `recordblocking(recorder,length)`
`resume`	Restart recording from paused position.
`set`	Set properties of `audiorecorder` object.
`stop`	Stop recording.

See the reference pages for `get`, `getaudiodata`, `play`, `record`, `recordblocking`, and `set` for additional syntax options.

Properties

`BitsPerSample`	Number of bits per sample. (Read-only)
`CurrentSample`	Current sample that the audio input device is recording. If the device is not recording, `CurrentSample` is the next sample to record with `record` or `resume`. (Read-only)
`DeviceID`	Identifier for audio device. (Read-only)
`NumberOfChannels`	Number of audio channels. (Read-only)
`Running`	Status of the audio recorder: `'on'` or `'off'`. (Read-only)
`SampleRate`	Sampling frequency in Hz. (Read-only)
`TotalSamples`	Total length of the audio data in samples. (Read-only)
`Tag`	String that labels the object.
`Type`	Name of the class: `'audiorecorder'`. (Read-only)
`UserData`	Any type of additional data to store with the object.

The following four properties apply to callback functions. The first two inputs to your callback function must be the `audiorecorder` object and an *`event`* structure.

`StartFcn`	Function to execute one time when recording starts.
`StopFcn`	Function to execute one time when recording stops.

`TimerFcn`	Function to execute repeatedly during recording. To specify time intervals for the repetitions, use the `TimerPeriod` property.
`TimerPeriod`	Time in seconds between `TimerFcn` callbacks. Default: `.05`

`audiorecorder` ignores any specified values for these properties, which will be removed in a future release.

`BufferLength`	Length of buffer in seconds.
`NumberOfBuffers`	Number of buffers.

Examples

Create an `audiorecorder` object for CD-quality audio in stereo, and view its properties:

```
recObj = audiorecorder(44100, 16, 2);
get(recObj)
```

Collect a sample of your speech with a microphone, and plot the signal data:

```
% Record your voice for 5 seconds.
recObj = audiorecorder;
disp('Start speaking.')
recordblocking(recObj, 5);
disp('End of Recording.');

% Play back the recording.
play(recObj);

% Store data in double-precision array.
myRecording = getaudiodata(recObj);

% Plot the waveform.
plot(myRecording);
```

See Also

`audiodevinfo` | `audioplayer` | `sound`

How To

- "Characteristics of Audio Files"
- "Record Audio"
- "Recording or Playing Audio within a Function"

Purpose

Information about NeXT/SUN (`.au`) sound file

Note `aufinfo` will be removed in a future release. Use `audioinfo` instead.

Syntax

```
[m d] = aufinfo(aufile)
```

Description

`[m d] = aufinfo(aufile)` returns information about the contents of the AU sound file specified by the string `aufile`.

`m` is the string `'Sound (AU) file'`, if `filename` is an AU file. Otherwise, it contains an empty string (`''`).

`d` is a string that reports the number of samples in the file and the number of channels of audio data. If `filename` is not an AU file, it contains the string `'Not an AU file'`.

See Also

`audioread` | `audioinfo`

auread

Purpose

Read NeXT/SUN (`.au`) sound file

Note `auread` will be removed in a future release. Use `audioread` instead.

Syntax

```
y = auread(aufile)
[y,Fs] = auread(aufile)
[y,Fs,nbits] = auread(aufile)
[___] = auread(aufile,N)
[___] = auread(aufile,[N1 N2])
siz = auread(aufile,'size')
```

Description

`y = auread(aufile)` loads a sound file specified by the string `aufile`, returning the sampled data in `y`. The `.au` extension is appended if no extension is given. Amplitude values are in the range `[-1,+1]`. `auread` supports multichannel data in the following formats:

- 8-bit mu-law
- 8-, 16-, and 32-bit linear
- Floating-point

`[y,Fs] = auread(aufile)` returns the sample rate (`Fs`) in Hertz used to encode the data in the file.

`[y,Fs,nbits] = auread(aufile)` returns the number of bits per sample (`nbits`).

`[___] = auread(aufile,N)` returns only the first `N` samples from each channel in the file.

`[___] = auread(aufile,[N1 N2])` returns only samples `N1` through `N2` from each channel in the file.

`siz = auread(aufile,'size')` returns the size of the audio data contained in the file in place of the actual audio data, returning the vector `siz = [samples channels]`.

Examples

Create a sound file from the example file `handel.mat`, and read portions of the file back into MATLAB:

```
% Create .au file in current folder.
load handel.mat

hfile = 'handel.au';
auwrite(y, Fs, hfile)
clear y Fs

% Read the data back into MATLAB, and listen to audio.
[y, Fs, nbits] = auread(hfile);
sound(y, Fs);

% Pause before next read and playback operation.
duration = numel(y) / Fs;
pause(duration + 2)

% Read and play only the first 2 seconds.
nsamples = 2 * Fs;
[y2, Fs] = auread(hfile, nsamples);
sound(y2, Fs);
pause(4)

% Read and play the middle third of the file.
sizeinfo = auread(hfile, 'size');

tot_samples = sizeinfo(1);
startpos = tot_samples / 3;
endpos = 2 * startpos;

[y3, Fs] = auread(hfile, [startpos endpos]);
sound(y3, Fs);
```

See Also

`audiowrite` | `audioinfo` | `audioplayer` | `audiorecorder` | `sound` | `audioread`

auwrite

Purpose

Write NeXT/SUN (`.au`) sound file

Note `auwrite` will be removed in a future release.

Syntax

```
auwrite(y,aufile)
auwrite(y,Fs,aufile)
auwrite(y,Fs,N,aufile)
auwrite(y,Fs,N,method,aufile)
```

Description

`auwrite(y,aufile)` writes a sound file specified by the string *`aufile`*. The data should be arranged with one channel per column. Amplitude values outside the range `[-1,+1]` are clipped prior to writing. `auwrite` supports multichannel data for 8-bit mu-law and 8- and 16-bit linear formats.

`auwrite(y,Fs,aufile)` specifies the sample rate of the data in Hertz.

`auwrite(y,Fs,N,aufile)` selects the number of bits in the encoder. Allowable settings are `N = 8` and `N = 16`.

`auwrite(y,Fs,N,method,aufile)` allows selection of the encoding method, which can be either `'mu'` or `'linear'`. Note that mu-law files must be 8-bit. By default, `method = 'mu'`.

See Also

`audioread` | `audiowrite`

Purpose

Create new Audio/Video Interleaved (AVI) file

Note `avifile` will be removed in a future release. Use `VideoWriter` instead.

Syntax

```
aviobj = avifile(filename)
avifile(filename, ParameterName, ParameterValue)
```

Description

`aviobj = avifile(filename)` creates an `avifile` object, giving it the name specified in *`filename`*, using default values for all `avifile` object properties. If *`filename`* does not include an extension, `avifile` appends `.avi` to the file name. AVI is a file format for storing audio and video data.

`avifile` returns a handle to an AVI file object *`aviobj`*. Use this object to refer to the AVI file in other functions. An AVI file object supports properties and methods that control aspects of the AVI file created.

`aviobj = avifile(filename, ParameterName, ParameterValue)` accepts one or more comma-separated parameter name/value pairs. Set parameter values before any calls to `addframe`. The following table lists the available parameters and values.

Parameter Name	Value	Default
`'colormap'`	An *m*-by-3 matrix defining the colormap for indexed AVI movies, where *m* is no more than 256 (236 for `Indeo` compression). Valid only when the `'compression'` is `'MSVC'`, `'RLE'`, or `'None'`.	No default
`'compression'`	A text string specifying the compression codec to use. To create an uncompressed file, specify a value of `'None'`.	`'Indeo5'` on Windows systems.

Parameter Name	Value	Default
	On UNIX® operating systems, the only valid value is `'None'`. On Windows systems, valid values include: • `'MSVC'` • `'RLE'` • `'Cinepak'` on 32-bit systems. • `'Indeo3'` or `'Indeo5'` on 32-bit Windows XP systems. Alternatively, specify a custom compression codec on Windows systems using the four-character code that identifies the codec (typically included in the codec documentation). If MATLAB cannot find the specified codec, it returns an error.	`'None'` on UNIX systems.
`'fps'`	A scalar value specifying the speed of the AVI movie in frames per second (fps).	15 fps
`'keyframe'`	For compressors that support temporal compression, the number of key frames per second.	2.1429 key frames per second
`'quality'`	A number from 0 through 100. Higher quality numbers result in higher video quality and larger file sizes. Lower quality numbers result in lower video quality and smaller file sizes. Valid only for compressed movies.	75
`'videoname'`	A descriptive name for the video stream, no more than 64 characters.	*`filename`*

Tips

- On some Windows systems, including all 64-bit systems, the default Indeo® 5 codec is not available. MATLAB issues a warning, and creates an uncompressed file.
- On 32-bit Windows XP systems, MATLAB can create AVI files compressed with Indeo 3 and Indeo 5 codecs. However, Microsoft Windows XP Service Pack 3 (SP3) with Security Update 954157 disables playback of Indeo 3 and Indeo 5 codecs in Windows Media Player and Internet Explorer®. Consider specifying a `compression` value of `'None'`.
- `avifile` cannot write files larger than 2 GB.
- You can use dot notation to set `avifile` object properties. For example, set the `quality` property to 100:

  ```
  aviobj = avifile('myavifile');
  aviobj.quality = 100;
  ```

  All property names of an `avifile` object are the same as the parameter names, except for the `keyframe` parameter, which corresponds to the `KeyFramePerSec` property. For example, change `keyframe` to `2.5`:

  ```
  aviobj.KeyFramePerSec = 2.5;
  ```

Examples

Create the AVI file `example.avi`:

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

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

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

Alternatives

Use `VideoWriter` rather than `avifile` to create AVI files. `VideoWriter` supports files larger than 2 GB, and by default, creates files with Motion JPEG compression, which all platforms support.

See Also

`VideoWriter` | `addframe (avifile)` | `close (avifile)` | `movie2avi`

Purpose

Information about Audio/Video Interleaved (AVI) file

Note `aviinfo` will be removed in a future release. Use `VideoReader` and the `get` method instead.

Syntax

```
fileinfo = aviinfo(filename)
```

Description

`fileinfo` `= aviinfo(`*`filename`*`)` returns a structure whose fields contain information about the AVI file specified in the string *`filename`*. If *`filename`* does not include an extension, then `.avi` is used. The file must be in the current working directory or in a directory on the MATLAB path.

The set of fields in the *`fileinfo`* structure is shown below.

Field Name	Description
`AudioFormat`	String containing the name of the format used to store the audio data, if audio data is present
`AudioRate`	Integer indicating the sample rate in Hertz of the audio stream, if audio data is present
`Filename`	String specifying the name of the file
`FileModDate`	String containing the modification date of the file
`FileSize`	Integer indicating the size of the file in bytes
`FramesPerSecond`	Integer indicating the desired frames per second
`Height`	Integer indicating the height of the AVI movie in pixels

Field Name	Description
`ImageType`	String indicating the type of image. Either `'truecolor'` for a truecolor (RGB) image, or `'indexed'` for an indexed image.
`NumAudioChannels`	Integer indicating the number of channels in the audio stream, if audio data is present
`NumFrames`	Integer indicating the total number of frames in the movie
`NumColormapEntries`	Integer specifying the number of colormap entries. For a truecolor image, this value is 0 (zero).
`Quality`	Number between 0 and 100 indicating the video quality in the AVI file. Higher quality numbers indicate higher video quality; lower quality numbers indicate lower video quality. This value is not always set in AVI files and therefore can be inaccurate.
`VideoCompression`	String containing the compressor used to compress the AVI file. If the compressor is not Microsoft Video 1, Run Length Encoding (RLE), Cinepak, or Intel® Indeo, `aviinfo` returns the four-character code that identifies the compressor.
`Width`	Integer indicating the width of the AVI movie in pixels

See also

`mmfileinfo`, `VideoReader`, `VideoWriter`

Purpose

Read Audio/Video Interleaved (AVI) file

Note `aviread` will be removed in a future release. Use `VideoReader` instead.

Syntax

```
mov = aviread(filename)
mov = aviread(filename, index)
```

Description

`mov = aviread(filename)` reads the AVI movie `filename` into the MATLAB movie structure `mov`. If `filename` does not include an extension, then `.avi` is used. Use the `movie` function to view the movie `mov`. On UNIX platforms, `filename` must be an uncompressed AVI file.

`mov` has two fields, `cdata` and `colormap`. The content of these fields varies depending on the type of image.

Image Type	cdata Field	colormap Field
`Truecolor`	Height-by-width-by-3 array of `uint8` values	Empty
`Indexed`	Height-by-width array of `uint8` values	`m`-by-3 array of `double` values

`aviread` supports 8-bit frames, for indexed and grayscale images, 16-bit grayscale images, or 24-bit truecolor images. Note, however, that `movie` only accepts 8-bit image frames; it does not accept 16-bit grayscale image frames.

`mov = aviread(filename, index)` reads only the frames specified by `index`. `index` can be a single index or an array of indices into the video stream. In AVI files, the first frame has the index value 1, the second frame has the index value 2, and so on.

See also

`mmfileinfo`, `movie`, `VideoReader`, `VideoWriter`

Purpose

Create axes graphics object

Syntax

```
axes
axes('PropertyName',propertyvalue,...)
axes(h)
h = axes(...)
```

Properties

For a list of properties, see Axes Properties.

Description

`axes` creates an axes graphics object in the current figure using default property values. `axes` is the low-level function for creating axes graphics objects. MATLAB automatically creates an axes, if one does not already exist, when you issue a command that creates a graph.

`axes('`*`PropertyName`*`',propertyvalue,...)` creates an axes object having the specified property values. For a description of the properties, see Axes Properties. MATLAB uses default values for any properties that you do not explicitly define as arguments. The `axes` function accepts property name/property value pairs, structure arrays, and cell arrays as input arguments (see the `set` and `get` commands for examples of how to specify these data types). While the basic purpose of an axes object is to provide a coordinate system for plotted data, axes properties provide considerable control over the way MATLAB displays data.

`axes(h)` makes existing axes `h` the current axes and brings the figure containing it into focus. It also makes `h` the first axes listed in the figure's `Children` property and sets the figure's `CurrentAxes` property to `h`. The current axes is the target for functions that draw image, line, patch, rectangle, surface, and text graphics objects.

If you want to make an axes the current axes without changing the state of the parent figure, set the `CurrentAxes` property of the figure containing the axes:

```
set(figure_handle,'CurrentAxes',axes_handle)
```

This command is useful if you want a figure to remain minimized or stacked below other figures, but want to specify the current axes.

`h = axes(...)` returns the handle of the created axes object.

Use the `set` function to modify the properties of an existing axes or the `get` function to query the current values of axes properties. Use the `gca` command to obtain the handle of the current axes.

The `axis` (not `axes`) function provides simplified access to commonly used properties that control the scaling and appearance of axes.

Set default axes properties on the figure and `root object` levels:

```
set(0,'DefaultAxesPropertyName',PropertyValue,...)
set(gcf,'DefaultAxesPropertyName',PropertyValue,...)
```

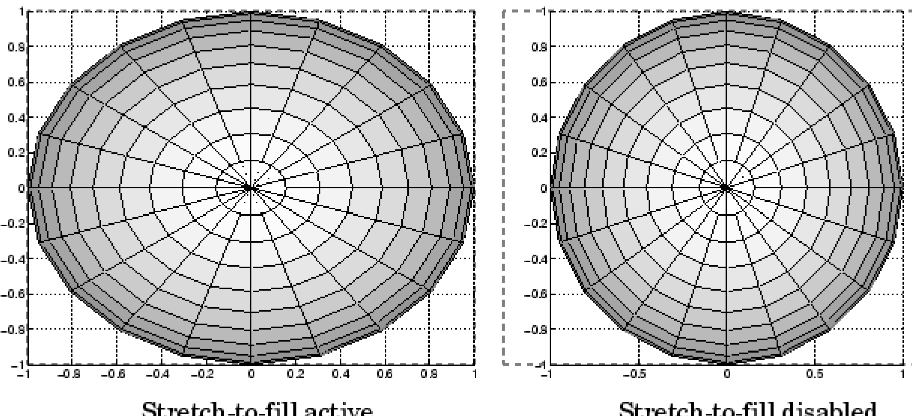
`PropertyName` is the name of the axes property and `PropertyValue` is the value you are specifying. Use `set` and `get` to access axes properties.

Stretch-to-Fill

By default, MATLAB stretches the axes to fill the axes position rectangle (the rectangle defined by the last two elements in the `Position` property). This results in graphs that use the available space in the rectangle. However, some 3-D graphs (such as a sphere) appear distorted because of this stretching, and are better viewed with a specific three-dimensional aspect ratio.

Stretch-to-fill is active when the `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, and `CameraViewAngleMode` are all auto (the default). However, stretch-to-fill is turned off when the `DataAspectRatio`, `PlotBoxAspectRatio`, or `CameraViewAngle` is user-specified, or when one or more of the corresponding modes is set to manual (which happens automatically when you set the corresponding property value).

This picture shows the same sphere displayed both with and without the stretch-to-fill. The dotted lines show the axes rectangle.



Stretch-to-fill active

Stretch-to-fill disabled

When stretch-to-fill is disabled, MATLAB sets the size of the axes to be as large as possible within the constraints imposed by the `Position` rectangle without introducing distortion. In the picture above, the height of the rectangle constrains the axes size.

Examples

Zoom in using aspect ratio and limits:

```
sphere
set(gca,'DataAspectRatio',[1 1 1],...
        'PlotBoxAspectRatio',[1 1 1],'ZLim',[-0.6 0.6])
```

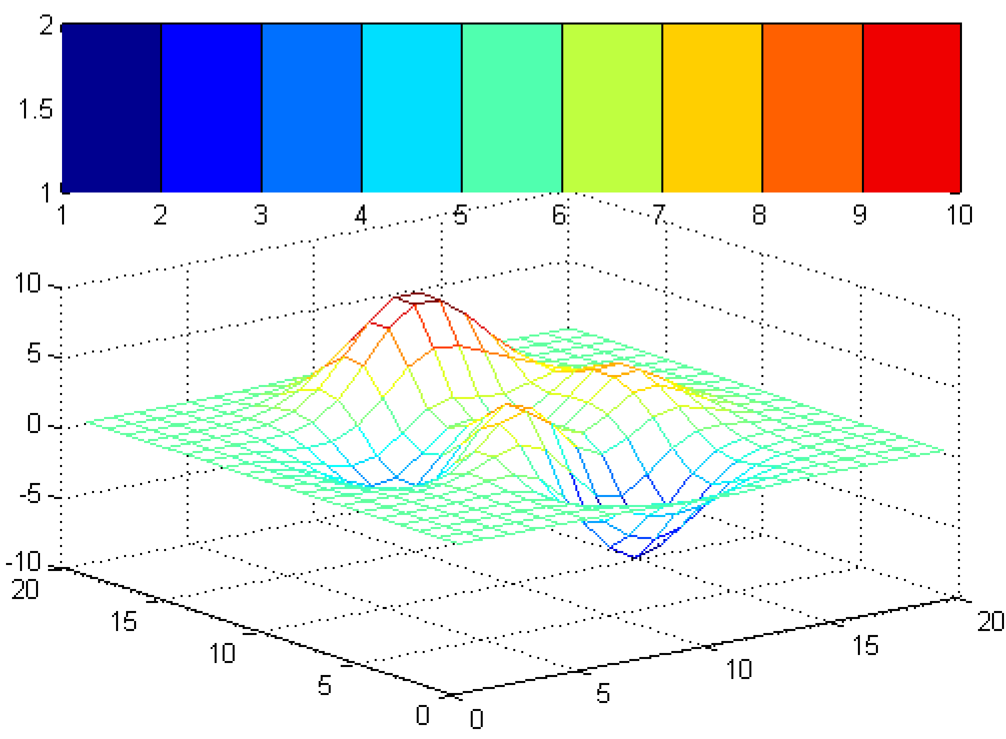
Zoom in and out using the `CameraViewAngle`:

```
sphere
set(gca,'CameraViewAngle',get(gca,'CameraViewAngle')-5)
set(gca,'CameraViewAngle',get(gca,'CameraViewAngle')+5)
```

Define multiple axes in a single figure window:

```
axes('position',[.1  .1  .8  .6])
mesh(peaks(20));
```

```
axes('position',[.1  .7  .8  .2])
pcolor([1:10;1:10]);
```



Alternatives

To create a figure select **New > Figure** from the figure window **File** menu. To add an axes to a figure, click one of the *New Subplots* icons in the Figure Palette, and slide right to select an arrangement of new axes. For details, see "Plotting Tools — Interactive Plotting".

See Also

`axis` | `cla` | `clf` | `figure` | `gca` | `grid` | `subplot` | `title` | `xlabel` | `ylabel` | `zlabel` | `view` | Axes Properties

Tutorials

- "Axes Objects — Defining Coordinate Systems for Graphs"

- "Axes Property Operations"

Purpose

Modify axes properties

Creating Axes Objects

Use `axes` to create axes objects.

Modifying Properties

You can set and query graphics object properties in two ways:

- "The Property Editor" is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands let you set and query the values of properties.

To change the default values of properties, see "Setting Default Property Values" in the Handle Graphics Objects documentation.

Axes Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`ActivePositionProperty`
`{outerposition} | position`

Use `OuterPosition` or `Position` property for resize. Specifies which property (`Position` or `OuterPosition`) MATLAB holds constant as you resize the figure (interactively or during a printing or exporting operation).

Setting a value for the `Position` property makes `position` the value of the `ActivePositionProperty`. The same is also true of setting a value for the `OuterPosition` property; `outerposition` becomes the value of the `ActivePositionProperty`.

See `OuterPosition` and `Position` for related properties.

See Automatic Axes Resize for a discussion of how to use axes positioning properties.

`ALim`
`[amin, amax]`

Alpha axis limits. Determines how MATLAB maps the `AlphaData` values of surface, patch, and image objects to the figure's alphamap.

- `amin` — Value of the data mapped to the first alpha value in the alphamap.
- `amax` — Value of the data mapped to the last alpha value in the alphamap.

MATLAB linearly interpolates data values in between across the alphamap and clamps data values outside to either the first or last alphamap value, whichever is closest.

If the axes contains multiple graphics objects, MATLAB sets `ALim` to span the range of all objects' `AlphaData` (or `FaceVertexAlphaData` for patch objects).

See the `alpha` function reference page for additional information.

`ALimMode`
`{auto} | manual`

Alpha axis limits mode.

- `auto` — MATLAB sets the `ALim` property to span the `AlphaData` limits of the graphics objects displayed in the axes.
- `manual` — MATLAB does not change the value of `ALim` when the `AlphaData` limits of axes children change.

Setting the `ALim` property sets `ALimMode` to `manual`.

`AmbientLightColor`
`ColorSpec`

Background light in a scene. Ambient light is a directionless light that shines uniformly on all objects in the axes. However, if there are no visible light objects in the axes, MATLAB does not use `AmbientLightColor`. If there are light objects in the axes, MATLAB adds the `AmbientLightColor` to the other light sources.

`AspectRatio`

(Obsolete)

This property produces a warning message when queried or changed. The `DataAspectRatio[Mode]` and `PlotBoxAspectRatio[Mode]` properties have superseded it.

`BeingDeleted`

`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

See the `close` and `delete` function reference pages for related information.

`Box`

`on | {off}`

Axes box mode. Specifies whether to enclose the axes extent in a box for 2-D views or a cube for 3-D views. The default is to not display the box.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Button press callback function. Executes whenever you press a mouse button while the pointer is within the axes, but not over another graphics object parented to the axes. For 3-D views, the active area is a rectangle that encloses the axes.

See the figure's `SelectionType` property to determine whether modifier keys were also pressed.

Set this property to a function handle that references the callback. The function must define at least two input arguments (handle of axes associated with the button down event and an event structure, which is empty for this property).

See Function Handle Callbacks for information on how to use function handles to define the callback function.

Some Plotting Functions Reset the ButtonDownFcn

Most MATLAB plotting functions clear the axes and reset a number of axes properties, including the `ButtonDownFcn` before plotting data. To create an interface that lets users plot data interactively, consider using a control device such as a push button (`uicontrol`), which plotting functions do not affect. See "Using Function Handles in GUIs" for an example.

If you must use the axes `ButtonDownFcn` to plot data, then you should use low-level functions such as `line`, `patch`, and `surface` and manage the process with the figure and axes `NextPlot` properties.

See "High-Level Versus Low-Level Functions" for information on how plotting functions behave.

See "Preparing Figures and Axes for Graphics" for more information.

Camera Properties

See View Control with the Camera Toolbar for information related to the Camera properties.

See Defining Scenes with Camera Graphics for information on the camera properties.

See View Projection Types for information on orthogonal and perspective projections.

`CameraPosition`

`[x, y, z]` axes coordinates

Location of the camera. Position from which the camera views the scene. Specify the point in axes coordinates.

If you fix `CameraViewAngle`, you can zoom in and out on the scene by changing the `CameraPosition`, moving the camera closer to the `CameraTarget` to zoom in and farther away from the `CameraTarget` to zoom out. As you change the `CameraPosition`, the amount of perspective also changes, if `Projection` is `perspective`. You can also zoom by changing the `CameraViewAngle`; however, this does not change the amount of perspective in the scene.

`CameraPositionMode`
`{auto} | manual`

Auto or manual `CameraPosition`. When set to `auto`, MATLAB automatically calculates the `CameraPosition` such that the camera lies a fixed distance from the `CameraTarget` along the azimuth and elevation specified by `view`. Setting a value for `CameraPosition` sets this property to manual.

`CameraTarget`
`[x, y, z]` axes coordinates

Camera aiming point. Specifies the location in the axes that the camera points to. The `CameraTarget` and the `CameraPosition` define the vector (the view axis) along which the camera looks.

`CameraTargetMode`
`{auto} | manual`

Auto or manual `CameraTarget` placement. When this property is `auto`, MATLAB automatically positions the `CameraTarget` at the centroid of the axes plot box. Specifying a value for `CameraTarget` sets this property to `manual`.

`CameraUpVector`
`[x, y, z]` axes coordinates

Camera rotation. Specifies the rotation of the camera around the viewing axis defined by the `CameraTarget` and the `CameraPosition` properties. Specify `CameraUpVector` as a

three-element array containing the *x*, *y*, and *z* components of the vector. For example, `[0 1 0]` specifies the positive *y*-axis as the up direction.

The default `CameraUpVector` is `[0 0 1]`, which defines the positive *z*-axis as the up direction.

`CameraUpVectorMode`
`{auto} | manual`

Default or user-specified up vector. When `CameraUpVectorMode` is `auto`, MATLAB uses a value of `[0 0 1]` (positive *z*-direction is up) for 3-D views and `[0 1 0]` (positive *y*-direction is up) for 2-D views. Setting a value for `CameraUpVector` sets this property to `manual`.

`CameraViewAngle`
scalar greater than 0 and less than or equal to 180 (angle in degrees)

Field of view. Determines the camera field of view. Changing this value affects the size of graphics objects displayed in the axes, but does not affect the degree of perspective distortion. The greater the angle, the larger the field of view, and the smaller objects appear in the scene.

`CameraViewAngleMode`
`{auto} | manual`

Auto or manual `CameraViewAngle`. When in `auto` mode, MATLAB sets `CameraViewAngle` to the minimum angle that captures the entire scene (up to 180°).

The following table summarizes MATLAB camera behavior using various combinations of `CameraViewAngleMode`, `CameraTargetMode`, and `CameraPositionMode`:

CameraViewAngleMode	CameraTargetMode	CameraPositionMode	Behavior
auto	auto	auto	Sets CameraTarget to plot box centroid, sets CameraViewAngle to capture entire scene, sets CameraPosition along the view axis.
auto	auto	manual	Sets CameraTarget to plot box centroid, sets CameraViewAngle to capture entire scene.
auto	manual	auto	Sets CameraViewAngle to capture entire scene, sets CameraPosition along the view axis.
auto	manual	manual	Sets CameraViewAngle to capture entire scene.
manual	auto	auto	Sets CameraTarget to plot box centroid, sets CameraPosition along the view axis.
manual	auto	manual	Sets CameraTarget to plot box centroid.

CameraViewAngleMode	CameraTargetMode	CameraPositionMode	Behavior
manual	manual	auto	Sets CameraPosition along the view axis.
manual	manual	manual	User specifies all camera properties.

Children
vector of graphics object handles

Handles of all graphics objects rendered within the axes (whether visible or not). The graphics objects that can be children of axes are image, light, line, patch, rectangle, surface, and text. Change the order of the handles to change the stacking of the objects on the display.

The text objects used to label the *x*-, *y*-, and *z*-axes and the title are also children of axes, but their HandleVisibility properties are set to off. This means their handles do not show up in the axes Children property unless you set the Root ShowHiddenHandles property to on.

When an object's HandleVisibility property is off, its parent's Children property does not list it. See HandleVisibility for more information.

CLim
[cmin, cmax]

Color axis limits. Determines how MATLAB maps the CData values of surface and patch objects to the figure's Colormap. cmin is the value of the data mapped to the first color in the colormap. cmax is the value of the data mapped to the last color in the colormap. MATLAB linearly interpolates data values in between across the colormap and clamps data values outside to either the first or last alphamap colormap color, whichever is closest.

When `CLimMode` is `auto` (the default), MATLAB assigns `cmin` the minimum data value and `cmax` the maximum data value in the graphics object's `CData`. This maps `CData` elements with minimum data value to the first `colormap` entry and with maximum data value to the last `colormap` entry.

If the axes contains multiple graphics objects, MATLAB sets `CLim` to span the range of all objects' `CData`.

See the `caxis` function reference page for related information.

`CLimMode`
: `{auto} | manual`

Color axis limits mode.

- `auto` — MATLAB sets the `CLim` property to span the `CData` limits of the graphics objects displayed in the axes.
- `manual` — MATLAB does not change the value of `CLim` when the `CData` limits of axes children change.

Setting the `CLim` property sets this property to `manual`.

`Clipping`
: `{on} | off`

Clipping mode. This property has no effect on axes.

`Color`
: `{[1,1,1]} | ColorSpec`

Color of the axes back planes. Setting this property to `none` means that the axes is transparent and the figure color shows through. A `ColorSpec` is a three-element RGB vector or one of the MATLAB predefined names.

`ColorOrder`
: `m`-by-`3` matrix of RGB values

Colors to use for multiline plots. Defines the colors used by the `plot` and `plot3` functions to color each line plotted. If you do not specify a line color with `plot` and `plot3`, these functions cycle through the `ColorOrder` property to obtain the color for each line plotted. To obtain the current `ColorOrder`, which might be set during startup, get the property value:

```
get(gca,'ColorOrder')
```

Note that if the axes `NextPlot` property is `replace` (the default), high-level functions like `plot` reset the `ColorOrder` property before determining the colors to use. If you want MATLAB to use a `ColorOrder` that is different from the default, set `NextPlot` to `replacechildren`. You can also specify your own default `ColorOrder`.

`CreateFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback function executed during object creation. Executes when MATLAB creates an axes object. You must define this property as a default value for axes. For example, the statement:

```
set(0,'DefaultAxesCreateFcn',@ax_create)
```

defines a default value on the `Root` level that sets axes properties whenever you (or MATLAB) create an axes.

```
function ax_create(src,evnt)
 set(src,'Color','b',...
 'XLim',[1 10],...
 'YLim',[0 100])
end
```

MATLAB executes this function after setting all properties for the axes. Setting the `CreateFcn` property on an existing axes object has no effect.

MATLAB passes the handle of the object whose `CreateFcn` is being executed as the first argument to the callback function and is also accessible through the Root `CallbackObject` property, which can be queried using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`CurrentPoint`
2-by-3 matrix

Location of last button click, in axes data units. Contains the coordinates of two points defined by the location of the pointer at the last mouse click. MATLAB returns the coordinates with respect to the requested axes.

Clicking Within the Axes — Orthogonal Projection

The two points lie on the line that is perpendicular to the plane of the screen and passes through the pointer. This is true for both 2-D and 3-D views.

The 3-D coordinates are the points, in the axes coordinate system, where this line intersects the front and back surfaces of the axes volume (which is defined by the axes x, y, and z limits).

The returned matrix is of the form:

$$\begin{bmatrix} x_{front} & y_{front} & z_{front} \\ x_{back} & y_{back} & z_{back} \end{bmatrix}$$

where *`front`* defines the point nearest to the camera position. Therefore, if the `CurrentPoint` property returns the `cp` matrix , then the first row,

```
cp(1,:)
```

specifies the point nearest the viewer and the second row,

```
cp(2,:)
```

specifies the point furthest from the viewer.

Clicking Outside the Axes — Orthogonal Projection

When you click outside the axes volume, but within the figure, the returned values are:

- Back point — a point in the plane of the camera target (which is perpendicular to the viewing axis).
- Front point — a point in the camera position plane (which is perpendicular to the viewing axis).

These points lie on a line that passes through the pointer and is perpendicular to the camera target and camera position planes.

Clicking Within the Axes — Perspective Projection

The values of the current point when using perspective project can be different from the same point in orthographic projection because the shape of the axes volume can be different.

Clicking Outside the Axes — Perspective Projection

Clicking outside of the axes volume returns the front point as the current camera position at all times. Only the back point updates with the coordinates of a point that lies on a line extending from the camera position through the pointer and intersecting the camera target at the point.

Related Information

See the figure `CurrentPoint` property for more information.

DataAspectRatio
[dx dy dz]

Relative scaling of data units. Controls the relative scaling of data units in the *x*, *y*, and *z* directions. For example, setting this property to [1 2 1] causes the length of one unit of data in the *x*-direction to be the same length as two units of data in the *y*-direction and one unit of data in the *z*-direction.

Note that the DataAspectRatio property interacts with the PlotBoxAspectRatio, XLimMode, YLimMode, and ZLimMode properties to control how MATLAB scales the *x*-, *y*-, and *z*-axis. Setting the DataAspectRatio disables the stretch-to-fill behavior if DataAspectRatioMode, PlotBoxAspectRatioMode, and CameraViewAngleMode are all auto.

The following table describes the interaction between properties when you disable stretch-to-fill behavior.

Interaction Between Properties With Stretch-to-Fill Disabled

X-, Y-, Z-LimitModes	DataAspectRatio	PlotBoxAspectRatio	Behavior
auto	auto	auto	Limits chosen to span data range in all dimensions.
auto	auto	manual	Limits chosen to span data range in all dimensions. MATLAB modifies DataAspectRatio to achieve the requested PlotBoxAspectRatio

X-, Y-, Z-LimitModes	DataAspectRatio	PlotBoxAspectRatio	Behavior
			within the limits the software selected.
`auto`	`manual`	`auto`	Limits chosen to span data range in all dimensions. MATLAB modifies `PlotBoxAspectRatio` to achieve the requested `DataAspectRatio` within the limits the software selected.
`auto`	`manual`	`manual`	Limits chosen to completely fit and center the plot within the requested `PlotBoxAspectRatio` given the requested `DataAspectRatio` (this might produce empty space around 2 of the 3 dimensions).
`manual`	`auto`	`auto`	MATLAB honors limits and modifies the `DataAspectRatio` and `PlotBoxAspectRatio` as necessary.

X-, Y-, Z-LimitModes	DataAspectRatio	PlotBoxAspectRatio	Behavior
`manual`	`auto`	`manual`	MATLAB honors limits and `PlotBoxAspectRatio` and modifies `DataAspectRatio` as necessary.
`manual`	`manual`	`auto`	MATLAB honors limits and `DataAspectRatio` and modifies the `PlotBoxAspectRatio` as necessary.
`1 manual` `2 auto`	`manual`	`manual`	MATLAB selects the 2 automatic limits to honor the specified aspect ratios and limit. See "Examples."
`2 or 3 manual`	`manual`	`manual`	MATLAB honors limits and `DataAspectRatio` while ignoring`PlotBoxAspectRatio`.

See "Understanding Axes Aspect Ratio" for more information.

`DataAspectRatioMode`
`{auto} | manual`

User or MATLAB controlled data scaling. Controls whether the values of the `DataAspectRatio` property are user-defined or selected automatically by MATLAB. Setting values for the `DataAspectRatio` property automatically sets this property

to `manual`. Changing `DataAspectRatioMode` to `manual` disables the stretch-to-fill behavior if `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, and `CameraViewAngleMode` are all auto.

`DeleteFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Delete axes callback function. Executes when you delete the axes object (for example, when you issue a `delete` or `clf` command). MATLAB executes the routine before destroying the object's properties so the callback can query these values.

MATLAB passes the handle of the object whose `DeleteFcn` is executing as the first argument to the callback function. The handle is also accessible through the Root `CallbackObject` property, which can be queried using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DrawMode`

`{normal} | fast`

Rendering mode. Controls the way MATLAB renders graphics objects displayed in the axes when the figure `Renderer` property is `painters`.

- `normal` — Draws objects in back to front ordering based on the current view in order to handle hidden surface elimination and object intersections.
- `fast` — Draws objects in the order in which you specify the drawing commands, without considering the relationships of the objects in three dimensions. This results in faster rendering because it requires no sorting of objects according to location in the view, but can produce undesirable results because it

bypasses the hidden surface elimination and object intersection handling provided by `normal DrawMode`.

When the figure `Renderer` property is `zbuffer`, MATLAB ignores `DrawMode` and always provides hidden surface elimination and object intersection handling.

`FontAngle`
`{normal} | italic | oblique`

Select italic or normal font. Selects the character slant for axes text. `normal` specifies a nonitalic font. `italic` and `oblique` specify italic font.

`FontName`
name (such as `Courier`) | `FixedWidth`

Font family name. Specifies the font to use for axes labels. To display and print properly, `FontName` must be a font that your system supports. Note that MATLAB does not display the *x*-, *y*-, and *z*-axis labels in a new font until you manually reset them (by setting the `XLabel`, `YLabel`, and `ZLabel` properties or by using the `xlabel`, `ylabel`, or `zlabel` command). Tick mark labels change immediately.

Specifying a Fixed-Width Font

If you want an axes to use a fixed-width font that looks good in any locale, set `FontName` to the string `FixedWidth`:

```
set(axes_handle,'FontName','FixedWidth')
```

This eliminates the need to hardcode the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding (such as in Japan, where character sets can be multibyte). A properly written MATLAB application that needs to use a fixed-width font should set `FontName` to `FixedWidth` (note that this string is case sensitive) and rely

on `FixedWidthFontName` to be set correctly in the end user's environment.

End users can adapt a MATLAB application to different locales or personal environments by setting the root `FixedWidthFontName` property to the appropriate value for that locale from `startup.m`.

Note that setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize`

Font size specified in `FontUnits`

Font size. An integer specifying the font size to use for axes labels and titles, in units determined by the `FontUnits` property. The default point size is `12` and the maximum allowable font size depends on your operating system. MATLAB does not display *x*-, *y*-, and *z*-axis text labels in a new font size until you manually reset them (by setting the `XLabel`, `YLabel`, or `ZLabel` properties or by using the `xlabel`, `ylabel`, or `zlabel` command). Tick mark labels change immediately.

`FontUnits`

`{points} | normalized | inches | centimeters | pixels`

Units used to interpret the `FontSize` property. When set to `normalized`, MATLAB interprets the value of `FontSize` as a fraction of the height of the axes. For example, a `normalized` `FontSize` of `0.1` sets the text characters to a font whose height is one tenth of the axes' height. The default units (`points`), are equal to 1/72 of an inch.

If you set both the `FontSize` and the `FontUnits` in one function call, you must set the `FontUnits` property first so that MATLAB correctly interprets the specified `FontSize`.

`FontWeight`

`{normal} | bold | light | demi`

Select bold or normal font. The character weight for axes text. MATLAB does not display the *x*-, *y*-, and *z*-axis text labels in bold until you manually reset them (by setting the `XLabel`, `YLabel`, and `ZLabel` properties or by using the `xlabel`, `ylabel`, or `zlabel` commands). Tick mark labels change immediately.

`GridLineStyle`

`- | --| {:} | -. | none`

Line style used to draw grid lines. The line style is a string consisting of a character, in quotes, specifying solid lines (-), dashed lines (--), dotted lines(:), or dash-dot lines (-.). The default grid line style is dotted. To turn on grid lines, use the `grid` command.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

- `on` — Handles are always visible.
- `callback` — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, functions that obtain handles by searching the object hierarchy or

querying handle properties cannot return it. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When you restrict a handle's visibility by using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the Root's `CurrentFigure` property, objects do not appear in the Root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

You can set the Root `ShowHiddenHandles` property to on to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

`HitTest`
: `{on} | off`

Selectable by mouse click. Determines if the axes can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click the axes. If `HitTest` is `off`, clicking the axes selects the object below it (which is usually the figure containing it).

`Interruptible`
: `off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the `Interruptible` property affects only the callbacks for the`ButtonDownFcn` property. A *running* callback is

the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both the callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted.

For more information, see "Control Callback Execution and Interruption".

`Layer`

`{bottom} | top`

Draw axis lines below or above graphics objects. Determines whether to draw axis lines and tick marks on top or below axes children objects for any 2-D view (for example, when you are looking along the *x*-, *y*-, or *z*-axis). Use this property to place grid lines and tick marks on top of images.

`LineStyleOrder`

`LineSpec {a solid line '-'}`

Order of line styles and markers used in a plot. Specifies which line styles and markers to use and in what order when creating multiple-line plots. For example:

```
set(gca,'LineStyleOrder', '-*|:|o')
```

sets `LineStyleOrder` to solid line with asterisk marker, dotted line, and hollow circle marker. The default is (-), which specifies a solid line for all data plotted. Alternatively, you can create a cell array of character strings to define the line styles:

```
set(gca,'LineStyleOrder',{'-*',':','o'})
```

MATLAB supports four line styles, which you can specify any number of times in any order. MATLAB cycles through the line styles only after using all colors defined by the `ColorOrder` property. For example, the first eight lines plotted use the different colors defined by `ColorOrder` with the first line style. MATLAB then cycles through the colors again, using the second line style specified, and so on.

You can also specify line style and color directly with the `plot` and `plot3` functions or by altering the properties of the `line` or `lineseries` objects after creating the graph.

High-Level Functions and LineStyleOrder

Note that, if the axes `NextPlot` property is `replace` (the default), high-level functions like `plot` reset the `LineStyleOrder` property before determining the line style to use. If you want MATLAB to use a `LineStyleOrder` that is different from the default, set `NextPlot` to `replacechildren`.

Specifying a Default LineStyleOrder

You can specify your own default `LineStyleOrder`. For example:

```
set(0,'DefaultAxesLineStyleOrder',{'-*',':','o'})
```

creates a default value for the axes `LineStyleOrder` that high-level plotting functions will not reset.

`LineWidth`

line width in points

Width of axis lines. Specifies the width, in points, of the *x*-, *y*-, and *z*-axis lines. The default line width is 0.5 points 1 point = $^{1}/_{72}$ inch.

`MinorGridLineStyle`

`- | --| {:} | -.  | none`

Line style used to draw minor grid lines. The line style is a string consisting of one or more characters, in quotes, specifying solid lines (-), dashed lines (--), dotted lines (:), or dash-dot lines (-.). The default minor grid line style is dotted. To turn on minor grid lines, use the `grid minor` command.

`NextPlot`

`add | {replace} | replacechildren`

Where to draw the next plot. Determines how high-level plotting functions draw into an existing axes.

- `add` — Use the existing axes to draw graphics objects.

- `replace` — Reset all axes properties except `Position` to their defaults and delete all axes children before displaying graphics (equivalent to `cla reset`).
- `replacechildren` — Remove all child objects, but do not reset axes properties (equivalent to `cla`).

The `newplot` function simplifies the use of the `NextPlot` property and is useful for functions that draw graphs using only low-level object creation routines. Note that figure graphics objects also have a `NextPlot` property.

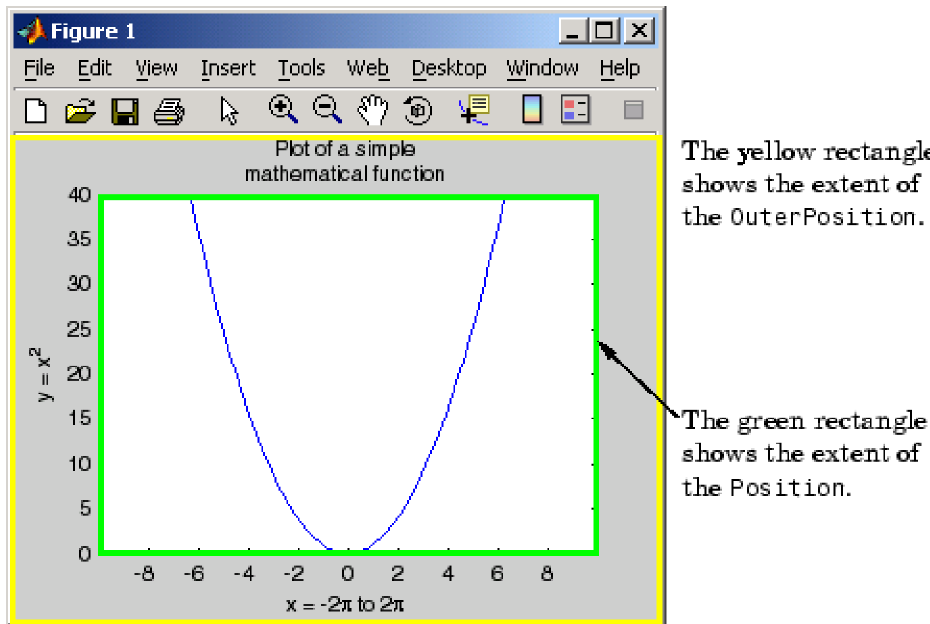
`OuterPosition`
four-element vector

Position of axes including labels, title, and a margin. Specifies a rectangle that locates the outer bounds of the axes, including axis labels, the title, and a margin. The vector is as follows:

```
[left bottom width height]
```

where `left` and `bottom` define the distance from the lower-left corner of the figure window to the lower-left corner of the rectangle. `width` and `height` are the dimensions of the rectangle.

The following picture shows the region defined by the `OuterPosition` enclosed in a yellow rectangle.



When ActivePositionProperty is OuterPosition (the default), resizing the figure will not clip any of the text. The default value of [0 0 1 1] (normalized units) includes the interior of the figure.

The Units property specifies all measurement units.

See the property for related information.

See "Automatic Axes Resize" for a discussion of how to use axes positioning properties.

Parent
figure or uipanel handle

Axes parent. The handle of the axes' parent object. The parent of an axes object is the figure which displays it or the uipanel object that contains it. The utility function gcf returns the handle of

the current axes `Parent`. You can reparent axes to other figure or uipanel objects.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`PlotBoxAspectRatio`
`[px py pz]`

Relative scaling of axes plot box. Controls the relative scaling of the plot box in the *x*, *y*, and *z* directions. The plot box is a box enclosing the axes data region as defined by the *x*-, *y*-, and *z*-axis limits.

Note that the `PlotBoxAspectRatio` property interacts with the `DataAspectRatio`, `XLimMode`, `YLimMode`, and `ZLimMode` properties to control the way MATLAB displays graphics objects. Setting the `PlotBoxAspectRatio` disables stretch-to-fill behavior, if `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, and `CameraViewAngleMode` are all `auto`.

`PlotBoxAspectRatioMode`
`{auto} | manual`

User or MATLAB controlled axis scaling. Controls whether the values of the `PlotBoxAspectRatio` property are user-defined or selected automatically by MATLAB. Setting values for the `PlotBoxAspectRatio` property automatically sets this property to `manual`. Changing the `PlotBoxAspectRatioMode` to `manual` disables stretch-to-fill behavior if `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, and `CameraViewAngleMode` are all `auto`.

`Position`
four-element vector

Position of axes. Specifies a rectangle that locates the axes within its parent container (figure or uipanel). The vector is of the form:

```
[left bottom width height]
```

where `left` and `bottom` define the distance from the lower-left corner of the container to the lower-left corner of the rectangle. `width` and `height` are the dimensions of the rectangle. The `Units` property specifies the units for all measurements.

When you enable axes stretch-to-fill behavior (when `DataAspectRatioMode`, `PlotBoxAspectRatioMode`, and `CameraViewAngleMode` are all `auto`), MATLAB stretches the axes to fill the `Position` rectangle. When you disable stretch-to-fill, MATLAB makes the axes as large as possible, while obeying all other properties, without extending outside the `Position` rectangle.

See the `OuterPosition` property for related information.

See "Automatic Axes Resize" for a discussion of how to use axes positioning properties.

`Projection`
`{orthographic} | perspective`

Type of projection. Selects one of the following projection types:

- `orthographic` — Maintains the correct relative dimensions of graphics objects regarding the distance a given point is from the viewer and draws parallel lines in the data parallel on the screen.
- `perspective` — Incorporates foreshortening, which allows you to perceive depth in 2-D representations of 3-D objects. Perspective projection does not preserve the relative dimensions of objects; it displays a distant line segment smaller than a nearer line segment of the same length. Parallel lines in the data might not appear parallel on screen.

`Selected`
`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection “handles” at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that the axes has been selected.

`SelectionHighlight`

`{on} | off`

Highlights objects when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`Tag`

string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

For example, suppose you want to direct all graphics output from a file to a particular axes, regardless of user actions that might have changed the current axes. To do this, identify the axes with a `Tag`:

```
axes('Tag','Special Axes')
```

Then make that axes the current axes before drawing by searching for the `Tag` with `findobj`:

```
axes(findobj('Tag','Special Axes'))
```

`TickDir`

`in | out`

Direction of tick marks. For 2-D views, the default is to direct tick marks inward from the axis lines; 3-D views direct tick marks outward from the axis line.

`TickDirMode`
`{auto} | manual`

Automatic tick direction control. In `auto` mode, MATLAB directs tick marks inward for 2-D views and outward for 3-D views. When you specify a setting for `TickDir`, MATLAB sets `TickDirMode` to `manual`. In `manual` mode, MATLAB does not change the specified tick direction.

`TickLength`
`[2DLength 3DLength]`

Length of tick marks. Specifies the length of axes tick marks. The first element is the length of tick marks used for 2-D views and the second element is the length of tick marks used for 3-D views. Specify tick mark lengths in units normalized relative to the longest of the visible *x*-, *y*-, or *z*-axis annotation lines.

`TightInset`
`[left bottom right top]` Read only

Margins added to Position to include text labels. The distances between the bounds of the `Position` property and the extent of the axes text labels and title. When added to the `Position` width and height values, the `TightInset` defines the tightest bounding box that encloses the axes and its labels and title.

See “Automatic Axes Resize” for more information.

`Title`
handle of text object

Axes title. The handle of the `text` object used for the axes title. Use this handle to change the properties of the title text or you can

set `Title` to the handle of an existing text object. For example, the following statement changes the color of the current title to red:

```
set(get(gca,'Title'),'Color','r')
```

To create a new title, set this property to the handle of the text object you want to use:

```
set(gca,'Title',text('String','New Title','Color','r'))
```

However, it is simpler to use the `title` command to create or replace an axes title:

```
title('New Title','Color','r') % Make text color red
title({'This title','has 2 lines'}) % Two line title
```

`Type`
: string (read-only)

Type of graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For axes objects, `Type` is always ’`axes`’.

`UIContextMenu`
: handle of uicontextmenu object

Associate a context menu with the axes. Assign this property the handle of a `uicontextmenu` object created in the axes’ parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the axes.

`Units`
: `inches | centimeters | {normalized} | points | pixels | characters`

Axes position units. The units used to interpret the Position property. MATLAB measures all units from the lower left corner of the figure window.

Note The Units property controls the positioning of the axes within the figure. This property does not affect the data units used for graphing. See the axes XLim, YLim, and ZLim properties to set the limits of each axis data units.

- normalized — Units map the lower left corner of the figure window to (0,0) and the upper right corner to (1.0, 1.0).
- inches, centimeters, and points — Absolute units. 1 point = $^1/_{72}$ inch.
- characters — Based on the size of characters in the default system font. The width of one characters unit is the width of the letter x, and the height of one characters unit is the distance between the baselines of two lines of text.

When specifying the units as property/value pairs during object creation, you must set the Units property before specifying the properties that you want to use these units.

UserData
matrix

User-specified data. Data you want to associate with the axes object. The default value is an empty array. MATLAB does not use this data, but you can access it using the set and get commands.

View
Obsolete

The axes camera properties now controls the functionality provided by the View property — CameraPosition, CameraTarget, CameraUpVector, and CameraViewAngle. See the view command.

`Visible`
`{on} | off`

Visibility of axes. By default, axes are visible. Setting this property to `off` prevents axis lines, tick marks, and labels from being displayed. The `Visible` property does not affect children of axes.

`XAxisLocation`
`top | {bottom}`

Location of x-axis tick marks and labels. Controls where MATLAB displays the *x*-axis tick marks and labels. Setting this property to `top` moves the *x*-axis to the top of the plot from its default position at the bottom. This property applies to 2–D views only.

`YAxisLocation`
`right | {left}`

Location of y-axis tick marks and labels. Controls where MATLAB displays the *y*-axis tick marks and labels. Setting this property to `right` moves the *y*-axis to the right side of the plot from its default position on the left side. This property applies to 2–D views only. See the `plotyy` function for a simple way to use two *y*-axes.

Properties That Control the X-, Y-, or Z-Axis

`XColor`
`YColor`
`ZColor`
`ColorSpec`

Color of axis lines. A three-element vector specifying an RGB triple, or a predefined MATLAB color string. This property determines the color of the axis lines, tick marks, tick mark labels, and the axis grid lines of the respective *x*-, *y*-, and *z*-axis. The default axis color is black. See `ColorSpec` for details on specifying colors.

`XDir`
`YDir`
`ZDir`

`{normal} | reverse`

Direction of increasing values. A mode controlling the direction of increasing axis values. Axes form a right-hand coordinate system. By default:

- *x*-axis values increase from left to right. To reverse the direction of increasing *x* values, set this property to `reverse`.

```
set(gca,'XDir','reverse')
```

- *y*-axis values increase from bottom to top (2-D view) or front to back (3-D view). To reverse the direction of increasing *y* values, set this property to `reverse`.

```
set(gca,'YDir','reverse')
```

- *z*-axis values increase pointing out of the screen (2-D view) or from bottom to top (3-D view). To reverse the direction of increasing *z* values, set this property to `reverse`.

```
set(gca,'ZDir','reverse')
```

`XGrid`
`YGrid`
`ZGrid`

`on | {off}`

Axis gridline mode. When you set any of these properties to `on`, MATLAB draws grid lines perpendicular to the respective axis (for example, along lines of constant *x*, *y*, or *z* values). Use the `grid` command to set all three properties `on` or `off` at once.

```
set(gca,'XGrid','on')
```

`XLabel`
`YLabel`
`ZLabel`

handle of text object

Axis labels. The handle of the `text` object used to label the *x*-, *y*-, or *z*-axis, respectively. To assign values to any of these properties, you must obtain the handle to the text string you want to use as a label. This statement defines a text object and assigns its handle to the `XLabel` property:

```
set(get(gca,'XLabel'),'String','axis label')
```

MATLAB places the string `'axis label'` appropriately for an *x*-axis label and moves any text object whose handle you specify as an `XLabel`, `YLabel`, or `ZLabel` property to the appropriate location for the respective label.

Alternatively, you can use the `xlabel`, `ylabel`, and `zlabel` functions, which generally provide a simpler means to label axis lines.

Note that using a bitmapped font (for example, Courier is usually a bitmapped font) might cause the labels to rotate improperly. As a workaround, use a TrueType font (for example, Courier New) for axis labels. See your system documentation to determine the types of fonts installed on your system.

`XLim`
`YLim`
`ZLim`

`[minimum maximum]`

Axis limits. Specifies the minimum and maximum values of the respective axis. The data you plot determines these values.

Changing these properties affects the scale of the *x*-, *y*-, or *z*-dimension as well as the placement of labels and tick marks on the axis. The default values for these properties are `[0 1]`.

See the `axis`, `datetick`, `xlim`, `ylim`, and `zlim` commands to set these properties.

`XLimMode`
`YLimMode`
`ZLimMode`
`{auto} | manual`

MATLAB or user-controlled limits. The axis limits mode determines whether MATLAB calculates axis limits based on the data plotted (for example, the `XData`, `YData`, or `ZData` of the axes children) or uses the values explicitly set with the `XLim`, `YLim`, or `ZLim` property, in which case, the respective limits mode is set to `manual`.

`XMinorGrid`
`YMinorGrid`
`ZMinorGrid`
`on | {off}`

Enable or disable minor gridlines. When set to `on`, MATLAB draws gridlines aligned with the minor tick marks of the respective axis. Note that you do not have to enable minor ticks to display minor grids.

`XMinorTick`
`YMinorTick`
`ZMinorTick`
`on | {off}`

Enable or disable minor tick marks. When set to `on`, MATLAB draws tick marks between the major tick marks of the respective axis. MATLAB automatically determines the number of minor ticks based on the space between the major ticks.

`XScale`
`YScale`
`ZScale`
`{linear} | log`

Axis scaling. Linear or logarithmic scaling for the respective axis. See also `loglog`, `semilogx`, and `semilogy`.

`XTick`
`YTick`
`ZTick`
vector of data values locating tick marks

Tick spacing. A vector of *x*-, *y*-, or *z*-data values that determine the location of tick marks along the respective axis. If you do not want tick marks displayed, set the respective property to the empty vector, `[ ]`. These vectors must contain monotonically increasing values.

`XTickLabel`
`YTickLabel`
`ZTickLabel`
string

Tick labels. A matrix of strings to use as labels for tick marks along the respective axis. These labels replace the numeric labels generated by MATLAB. If you do not specify enough text labels for all the tick marks, MATLAB uses all of the labels specified, then reuses the specified labels.

For example, the statement:

```
set(gca,'XTickLabel',{'One';'Two';'Three';'Four'})
```

labels the first four tick marks on the *x*-axis and then reuses the labels for the remaining ticks.

Labels can be cell arrays of strings, padded string matrices, string vectors separated by vertical slash characters, or numeric vectors (where MATLAB implicitly converts each number to the equivalent string using `num2str`). All of the following are equivalent:

```
set(gca,'XTickLabel',{'1';'10';'100'})
set(gca,'XTickLabel','1|10|100')
set(gca,'XTickLabel',[1;10;100])
set(gca,'XTickLabel',['1  ';'10 ';'100'])
```

Note that tick labels do not interpret TeX character sequences (however, the `Title`, `XLabel`, `YLabel`, and `ZLabel` properties do).

`XTickMode`
`YTickMode`
`ZTickMode`

`{auto} | manual`

MATLAB or user-controlled tick spacing. The axis tick modes determine whether MATLAB calculates the tick mark spacing based on the range of data for the respective axis (`auto` mode) or uses the values explicitly set for any of the `XTick`, `YTick`, and `ZTick` properties (`manual` mode). Setting values for the `XTick`, `YTick`, or `ZTick` properties sets the respective axis tick mode to `manual`.

`XTickLabelMode`
`YTickLabelMode`
`ZTickLabelMode`

`{auto} | manual`

MATLAB or user-determined tick labels. The axis tick mark labeling modes determine whether MATLAB uses numeric tick mark labels that span the range of the plotted data (`auto` mode) or uses the tick mark labels specified with the `XTickLabel`, `YTickLabel`, or `ZTickLabel` property (`manual` mode). Setting

values for the `XTickLabel`, `YTickLabel`, or `ZTickLabel` property sets the respective axis tick label mode to `manual`.

See Also

`axes`

Purpose

Axis scaling and appearance

Syntax

```
axis([xmin xmax ymin ymax])
axis([xmin xmax ymin ymax zmin zmax cmin cmax])
v = axis
axis auto
axis manual
axis tight
axis fill
axis ij
axis xy
axis equal
axis image
axis square
axis vis3d
axis normal
axis off
axis on
axis(axes_handles,...)
[mode,visibility,direction] = axis('state')
```

Description

`axis` manipulates commonly used axes properties. (See Algorithm section.)

`axis([xmin xmax ymin ymax])` sets the limits for the *x*- and *y*-axis of the current axes.

`axis([xmin xmax ymin ymax zmin zmax cmin cmax])` sets the *x*-, *y*-, and *z*-axis limits and the color scaling limits (see `caxis`) of the current axes.

`v = axis` returns a row vector containing scaling factors for the *x*-, *y*-, and *z*-axis. `v` has four or six components depending on whether the current axes is 2-D or 3-D, respectively. The returned values are the current axes `XLim`, `Ylim`, and `ZLim` properties.

`axis auto` sets MATLAB default behavior to compute the current axes limits automatically, based on the minimum and maximum values of *x*, *y*, and *z* data. You can restrict this automatic behavior to a specific

axis. For example, `axis 'auto x'` computes only the *x*-axis limits automatically; `axis 'auto yz'` computes the *y*- and *z*-axis limits automatically.

`axis manual` and `axis(axis)` freezes the scaling at the current limits, so that if `hold` is `on`, subsequent plots use the same limits. This sets the `XLimMode`, `YLimMode`, and `ZLimMode` properties to `manual`.

`axis tight` sets the axis limits to the range of the data.

`axis fill` sets the axis limits and `PlotBoxAspectRatio` so that the axes fill the position rectangle. This option has an effect only if `PlotBoxAspectRatioMode` or `DataAspectRatioMode` is `manual`.

`axis ij` places the coordinate system origin in the upper left corner. The *i*-axis is vertical, with values increasing from top to bottom. The *j*-axis is horizontal with values increasing from left to right.

`axis xy` draws the graph in the default Cartesian axes format with the coordinate system origin in the lower left corner. The *x*-axis is horizontal with values increasing from left to right. The *y*-axis is vertical with values increasing from bottom to top.

`axis equal` sets the aspect ratio so that the data units are the same in every direction. The aspect ratio of the *x*-, *y*-, and *z*-axis is adjusted automatically according to the range of data units in the *x*, *y*, and *z* directions.

`axis image` is the same as `axis equal` except that the plot box fits tightly around the data.

`axis square` makes the current axes region square (or cubed when three-dimensional). This option adjusts the *x*-axis, *y*-axis, and *z*-axis so that they have equal lengths and adjusts the increments between data units accordingly.

`axis vis3d` freezes aspect ratio properties to enable rotation of 3-D objects and overrides stretch-to-fill.

`axis normal` automatically adjusts the aspect ratio of the axes and the relative scaling of the data units so that the plot fits the figure's shape as well as possible.

`axis off` turns off all axis lines, tick marks, and labels.

`axis on` turns on all axis lines, tick marks, and labels.

`axis(axes_handles,...)` applies the `axis` command to the specified axes. For example, the following statements

```
h1 = subplot(221);
h2 = subplot(222);
axis([h1 h2],'square')
```

set both axes to `square`.

`[mode,visibility,direction] = axis('state')` returns three strings indicating the current setting of axes properties:

Output Argument	Strings Returned
mode	'auto' | 'manual'
visibility	'on' | 'off'
direction	'xy' | 'ij'

`mode` is `auto` if `XLimMode`, `YLimMode`, and `ZLimMode` are all set to `auto`. If `XLimMode`, `YLimMode`, or `ZLimMode` is `manual`, `mode` is `manual`.

Keywords to `axis` can be combined, separated by a space (e.g., `axis tight equal`). These are evaluated from left to right, so subsequent keywords can overwrite properties set by prior ones.

Tips

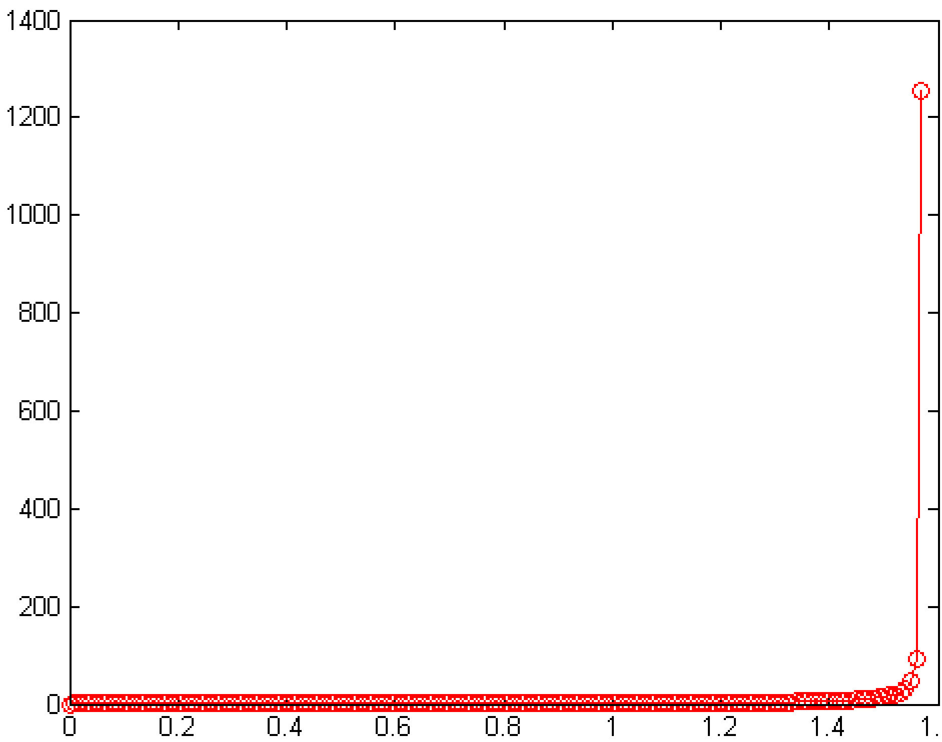
You can create an axes (and a figure for it) if none exists with the `axis` command. However, if you specify non-default limits or formatting for the axes when doing this, such as `[4 8 2 9]`, `square`, `equal`, or `image`, the property is ignored because there are no axis limits to adjust in the absence of plotted data. To use `axis` in this manner, you can set `hold on` to keep preset axes limits from being overridden.

Examples

The statements

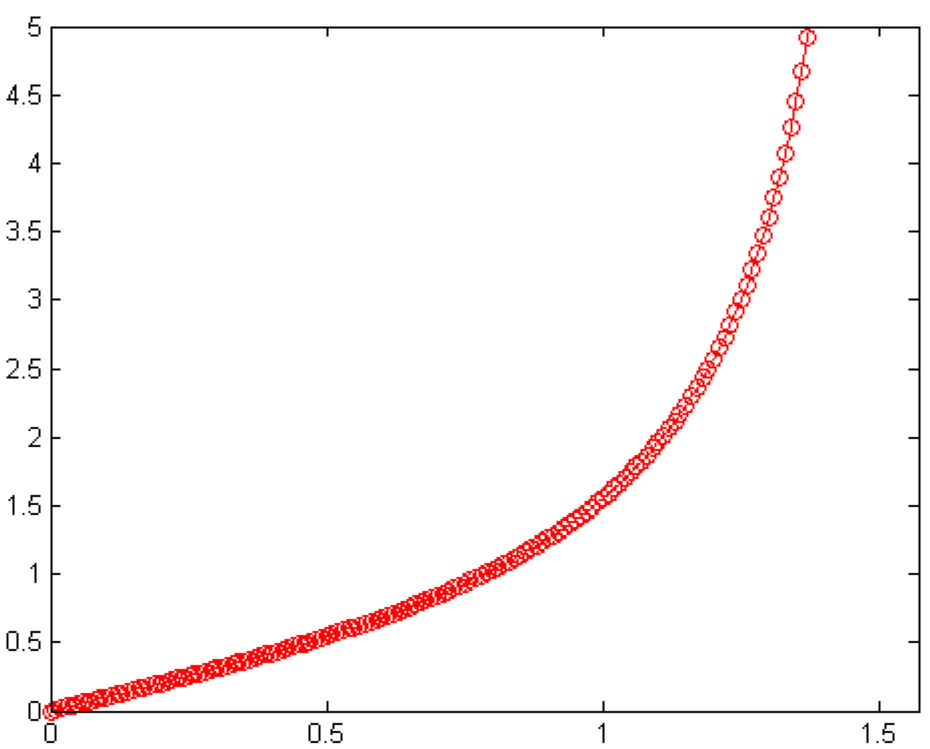
```
x = 0:.01:pi/2;
plot(x,tan(x),'-ro')
```

use the automatic scaling of the *y*-axis based on `ymax = tan(1.57)`, which is well over 1000:



The following figure shows a more satisfactory plot after typing

```
axis([0 pi/2 0 5])
```



Algorithms

When you specify minimum and maximum values for the *x*-, *y*-, and *z*-axes, `axis` sets the `XLim`, `Ylim`, and `ZLim` properties for the current axes to the respective minimum and maximum values in the argument list. Additionally, the `XLimMode`, `YLimMode`, and `ZLimMode` properties for the current axes are set to `manual`.

`axis auto` sets the current axes `XLimMode`, `YLimMode`, and `ZLimMode` properties to `'auto'`.

axis manual sets the current axes XLimMode, YLimMode, and ZLimMode properties to 'manual'.

The following table shows the values of the axes properties set by axis equal, axis normal, axis square, and axis image.

Axes Property or Behavior	axis equal	axis normal	axis square	axis image
DataAspectRatio property	[1 1 1]	not set	not set	[1 1 1]
DataAspectRatioMode property	manual	auto	auto	manual
PlotBoxAspectRatio property	[3 4 4]	not set	[1 1 1]	auto
PlotBoxAspectRatioMode property	manual	auto	manual	auto
Stretch-to-fill behavior;	disabled	active	disabled	disabled

See Also

axes | grid | subplot | xlim | ylim | zlim | axes properties

balance

Purpose

Diagonal scaling to improve eigenvalue accuracy

Syntax

```
[T,B] = balance(A)
[S,P,B] = balance(A)
B = balance(A)
B = balance(A,'noperm')
```

Description

`[T,B] = balance(A)` returns a similarity transformation `T` such that `B = T\A*T`, and `B` has, as nearly as possible, approximately equal row and column norms. `T` is a permutation of a diagonal matrix whose elements are integer powers of two to prevent the introduction of roundoff error. If `A` is symmetric, then `B == A` and `T` is the identity matrix.

`[S,P,B] = balance(A)` returns the scaling vector `S` and the permutation vector `P` separately. The transformation `T` and balanced matrix `B` are obtained from `A`, `S`, and `P` by `T(:,P) = diag(S)` and `B(P,P) = diag(1./S)*A*diag(S)`.

`B = balance(A)` returns just the balanced matrix `B`.

`B = balance(A,'noperm')` scales `A` without permuting its rows and columns.

Tips

- Nonsymmetric matrices can have poorly conditioned eigenvalues. Small perturbations in the matrix, such as roundoff errors, can lead to large perturbations in the eigenvalues. The condition number of the eigenvector matrix,

  ```
  cond(V) = norm(V)*norm(inv(V))
  ```

  where

  ```
  [V,T] = eig(A)
  ```

relates the size of the matrix perturbation to the size of the eigenvalue perturbation. Note that the condition number of A itself is irrelevant to the eigenvalue problem.

Balancing is an attempt to concentrate any ill conditioning of the eigenvector matrix into a diagonal scaling. Balancing usually cannot turn a nonsymmetric matrix into a symmetric matrix; it only attempts to make the norm of each row equal to the norm of the corresponding column.

Note The MATLAB eigenvalue function, `eig(A)`, automatically balances `A` before computing its eigenvalues. Turn off the balancing with `eig(A,'nobalance')`.

Examples

This example shows the basic idea. The matrix `A` has large elements in the upper right and small elements in the lower left. It is far from being symmetric.

```
A = [1  100  10000; .01  1  100; .0001  .01  1]
A =
   1.0e+04 *
    0.0001    0.0100    1.0000
    0.0000    0.0001    0.0100
    0.0000    0.0000    0.0001
```

Balancing produces a diagonal matrix `T` with elements that are powers of two and a balanced matrix `B` that is closer to symmetric than `A`.

```
[T,B] = balance(A)
T =
   1.0e+03 *
    2.0480         0         0
         0    0.0320         0
         0         0    0.0003
B =
    1.0000    1.5625    1.2207
```

```
    0.6400    1.0000    0.7813
    0.8192    1.2800    1.0000
```

To see the effect on eigenvectors, first compute the eigenvectors of `A`, shown here as the columns of `V`.

```
[V,E] = eig(A); V
V =
0.9999        -0.9999                -0.9999
0.0100         0.0059 + 0.0085i       0.0059 - 0.0085i
0.0001         0.0000 - 0.0001i       0.0000 + 0.0001i
```

Note that all three vectors have the first component the largest. This indicates `V` is badly conditioned; in fact `cond(V)` is `8.7766e+003`. Next, look at the eigenvectors of `B`.

```
[V,E] = eig(B); V
V =
0.6933        -0.6993                -0.6993
0.4437         0.2619 + 0.3825i       0.2619 - 0.3825i
0.5679         0.2376 - 0.4896i       0.2376 + 0.4896i
```

Now the eigenvectors are well behaved and `cond(V)` is `1.4421`. The ill conditioning is concentrated in the scaling matrix; `cond(T)` is `8192`.

This example is small and not really badly scaled, so the computed eigenvalues of `A` and `B` agree within roundoff error; balancing has little effect on the computed results.

Limitations

Balancing can destroy the properties of certain matrices; use it with some care. If a matrix contains small elements that are due to roundoff error, balancing might scale them up to make them as significant as the other elements of the original matrix.

See Also

`eig`

Purpose

Bar graph

Syntax

```
bar(Y)
bar(x,Y)

bar(___,width)
bar(___,style)
bar(___,bar_color)
bar(___,Name,Value)

bar(axes_handle,___)

h = bar(___)
```

Description

`bar(Y)` draws one bar for each element in `Y`.

`bar(x,Y)` draws bars for each column in `Y` at locations specified in `x`.

`bar(___,width)` sets the relative bar width and controls the separation of bars within a group and can include any of the input arguments in previous syntaxes.

`bar(___,style)` specifies the style of the bars and can include any of the input arguments in previous syntaxes.

`bar(___,bar_color)` displays all bars using the color specified by the single-letter abbreviation of `bar_color` and can include any of the input arguments in previous syntaxes.

`bar(___,Name,Value)` sets the property names to the specified values and can include any of the input arguments in previous syntaxes.

Note You cannot specify names and values when using `hist` or `histc` options.

`bar(axes_handle,___)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = bar(___)` returns a vector of handles to barseries graphics objects, one for each created.

Change the colormap to use a different color scheme. Use the `colormap` function to specify the figure colormap.

Use `shading flat` to turn off bar edges. Use `shading faceted` to turn on bar edges. See the `shading` function for more information.

Input Arguments

x - x-axis intervals

vector | 2D array

x-axis intervals for vertical bars, specified as a vector.

The *x*-values can be nonmonotonic, but cannot contain duplicate values.

Example: `x = 1:10;`

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Y - Bar lengths

vector | 2–D array

Bar lengths, specified as an array.

- If `Y` is a vector, `bar` creates `length(Y)` bars.
- If `Y` is a 2–D array, `bar` groups the bars produced by the elements in each row.

Example: `y = [10,8,5,7,3,9,1];`

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

width - Bar width

0.8 (default) | scalar

Bar width of each bar, specified as a fraction of the total width of a bar and the space between bars. The default of 0.8 means the a bar width is 80% of the space from the previous bar to the next bar, with 10% of that space on each side.

If `width` is `1`, the bars within a group touch one another.

Example: `bar(y,0.5);`

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

style - Bar style, specified as one of the following strings

`'grouped'` (default) | `'stacked'` | `'hist'` | `'histc'`

Bar style, specified by one of these values.

Style	Purpose
`'grouped'`	Displays *m* groups of *n* vertical bars, where *m* is the number of rows and *n* is the number of columns in `Y`. The group contains one bar per column in `Y`.
`'stacked'`	Displays one bar for each row in `Y`. • If `Y` is a vector, bar ignores the `` `stacked' `` option. • If `Y` is a matrix, the bar height is the sum of the elements in the

Style	Purpose
	row. Each bar is multicolored. Colors correspond to distinct elements and show the relative contribution each row element makes to the total sum.
`'histc'`	Displays the graph in histogram format, in which bars touch one another.
`'hist'`	Displays the graph in histogram format, but centers each bar over the *x*-ticks, rather than making bars span *x*-ticks as the `histc` option does.

Example: `bar(Y,'hist')`

Data Types
`char`

bar_color - Bar color
`'b' | 'r' | 'g' | 'b' | 'c' | 'm' | 'y' | 'k' | 'w'`

Bar color, specified as a single-letter abbreviation, assigns the color specified to all bars.

Example: `bar(Y,'r')`

Data Types
`char`

axes_handle - Axes handle
Current axes (default) | Axes handle

Axes handle specifying the target axes for the bar graph.

Example: `bar(ah,Y);`

Name-Value Pair Arguments

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

Example: `bar(Y,'EdgeColor','g')`

'BarLayout' - Arrangement of bars

`grouped` (default) | `stacked`

Arrangement of bars, specified as the comma-separated pair consisting of `'BarLayout'` and one of these values:

- `grouped` — Display *m* groups of *n* vertical bars, where *m* is the number of rows and *n* is the number of columns in the input argument `Y`. The group contains one bar per column in `Y`.
- `stacked` — Display one bar for each row in the input argument `Y`. The bar height is the sum of the elements in the row. Each bar is multicolored, with colors corresponding to distinct elements and showing the relative contribution each row element makes to the total sum.

'BaseValue' - Base line location

0 (default) | scalar

Base line location, specified as the comma-separated pair consisting of `'BaseValue'` and the value along the *y*-axis (vertical bars) or *x*-axis (horizontal bars) at which MATLAB draws the baseline.

'BarWidth' - Relative width of bars

0.8 (default) | scalar

Relative bar width, specified as the comma-separated pair consisting of `'BarWidth'` and a value relative to width and spacing. `BarWidth` controls the separation of bars within a group. By default, the bars

within a group have a slight separation. A value of `1` makes bars touch with no space. A value greater than `1` makes the bars overlap.

'EdgeColor' - Color of bar edges

`[0,0,0]` (default) | `none` | `ColorSpec`

Color of bar edges, specified as the comma-separated pair consisting of `'EdgeColor'` and a three-element RGB vector, or one of the MATLAB predefined color specifier names.

'FaceColor' - Color of bars

`flat` (default) | `none` | `ColorSpec`

Color of bars, specified as the comma-separated pair consisting of `'FaceColor'` and one of these values:

- `flat` — Use the figure `colormap` to determine the color of the bars.
- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for all bars.
- `none` — Do not draw faces. Note that MATLAB draws `EdgeColor` independently of `FaceColor`.

'LineStyle' - Line style of bar edges

– (default) | `-` | `--` | `:` | `-.` | `none`

Line style of bar edges, specified as the comma-separated pair consisting of `'LineStyle'` and one of the predefined line style strings.

'LineWidth' - Width of bar edges

0.5 points (default) | Scalar width specified in points

Width of bar edges, specified as the comma-separated pair consisting of `'LineWidth'` and a scalar value representing the line width in points.

Output Arguments

h - Handle to barseries objects, returned as a barseries handle

Array of one or more handles

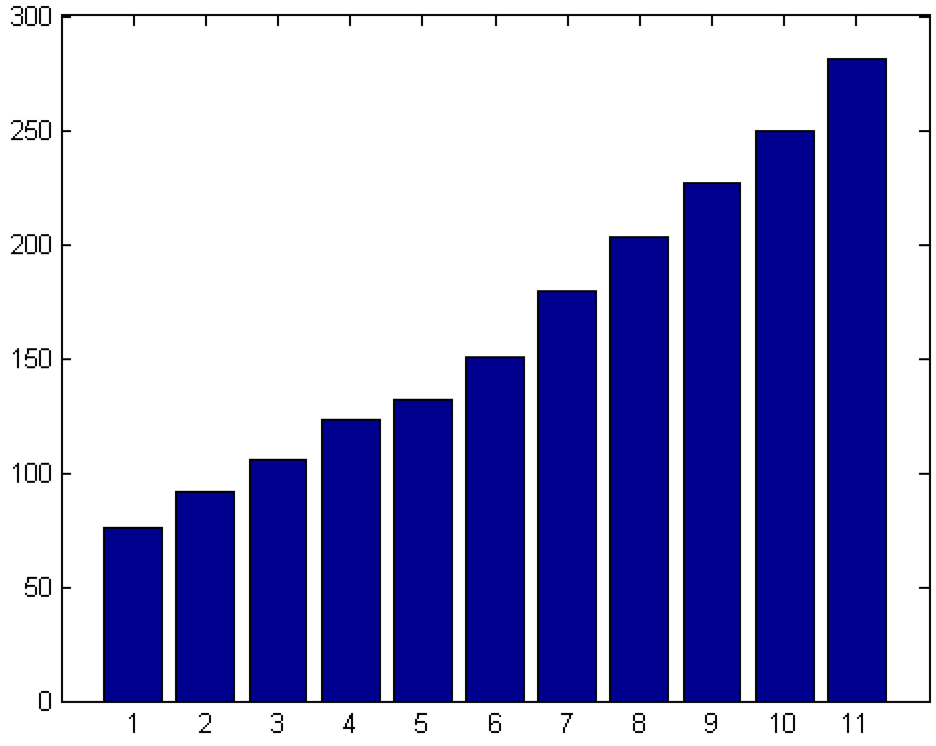
Handles to barseries objects created for the graph.

Examples

Single Data Series

Use the bar function to plot vector data.

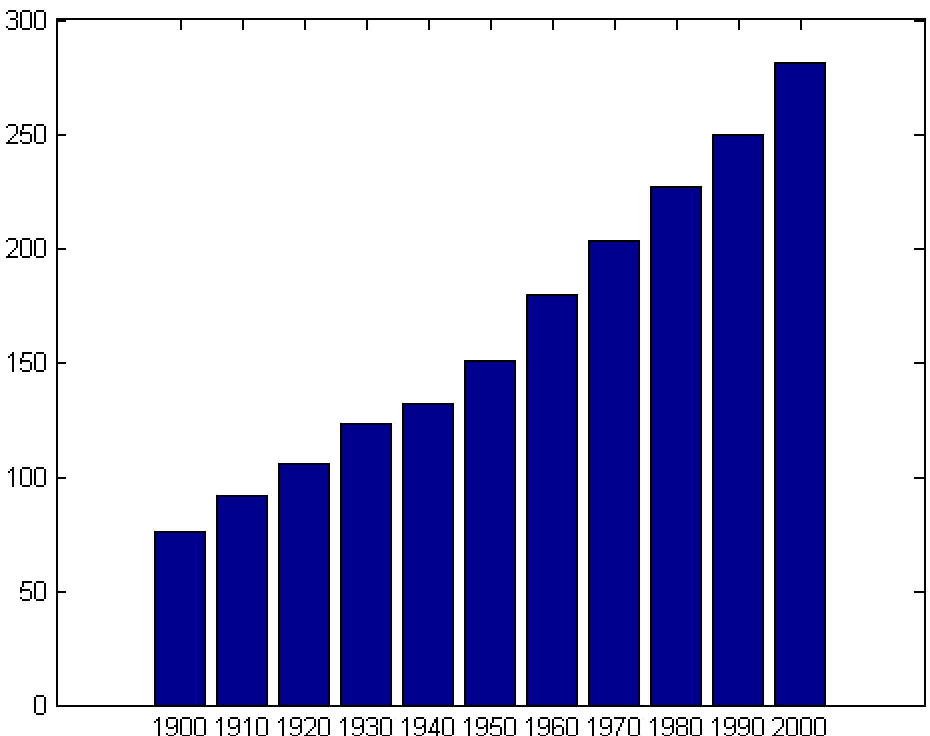
```
y = [75.995,91.972,105.711,123.203,131.669,...
     150.697,179.323,203.212,226.505,249.633,281.422];
figure; bar(y);
```



X-Axis Tick Labels

Specify the numeric values of the x-axis tick labels.

```
x = 1900:10:2000;
y = [75.995,91.972,105.711,123.203,131.669,...
     150.697,179.323,203.212,226.505,249.633,281.422];
figure; bar(x,y);
```

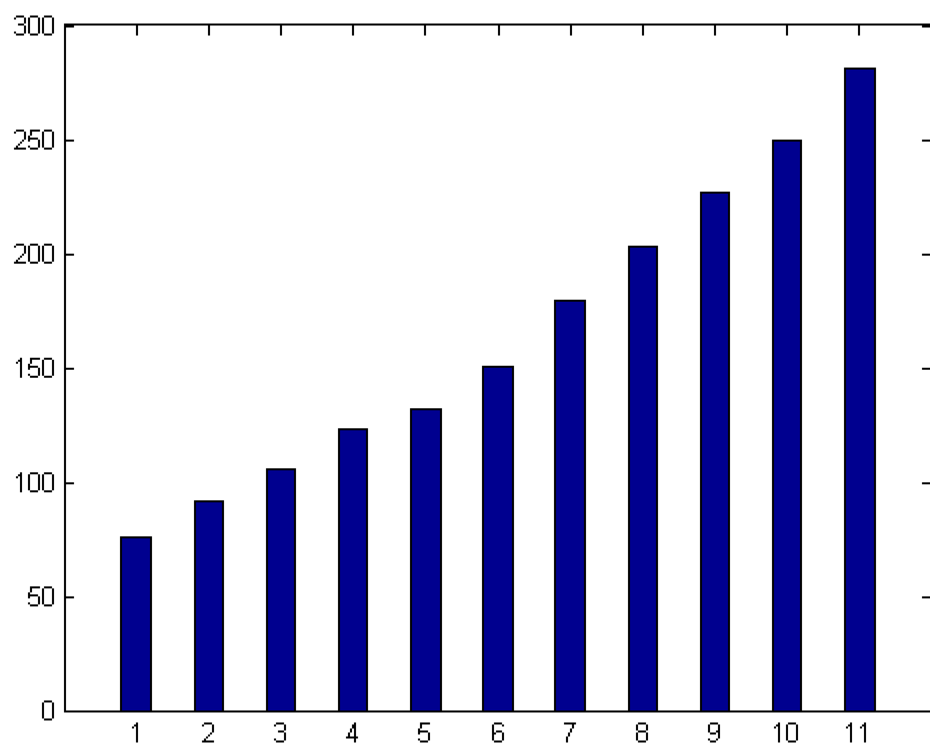


Plot *y* values at each *x* value. Notice that the `length(x)` and `length(y)` have to be same.

Bar Width

Set width of each bar to 40 percent of the total space available for each bar.

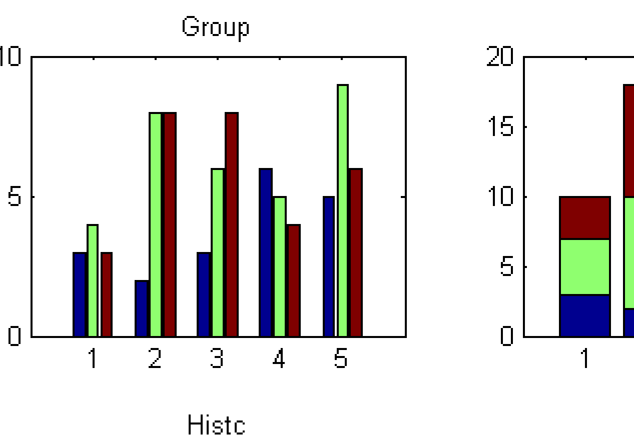
```
y = [75.995,91.972,105.711,123.203,131.669,...
     150.697,179.323,203.212,226.505,249.633,281.422];
figure; bar(y,0.4);
```

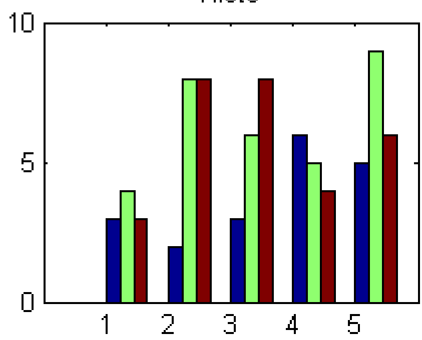


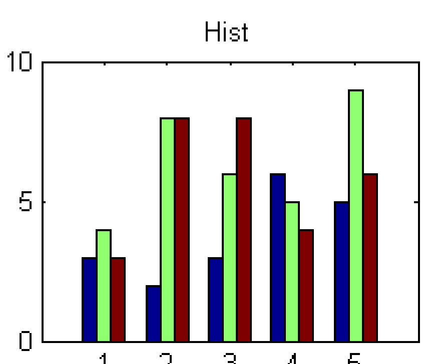
Bar Graph Styles

Create a 2x2 subplot with different styles applied to each bar graph.

```
Y = round(rand(5,3)*10);
figure;
subplot(2,2,1); bar(Y,'grouped'); title('Group')
subplot(2,2,2); bar(Y,'stacked'); title('Stack')
subplot(2,2,3); bar(Y,'histc'); title('Histc')
subplot(2,2,4); bar(Y,'hist'); title('Hist')
```



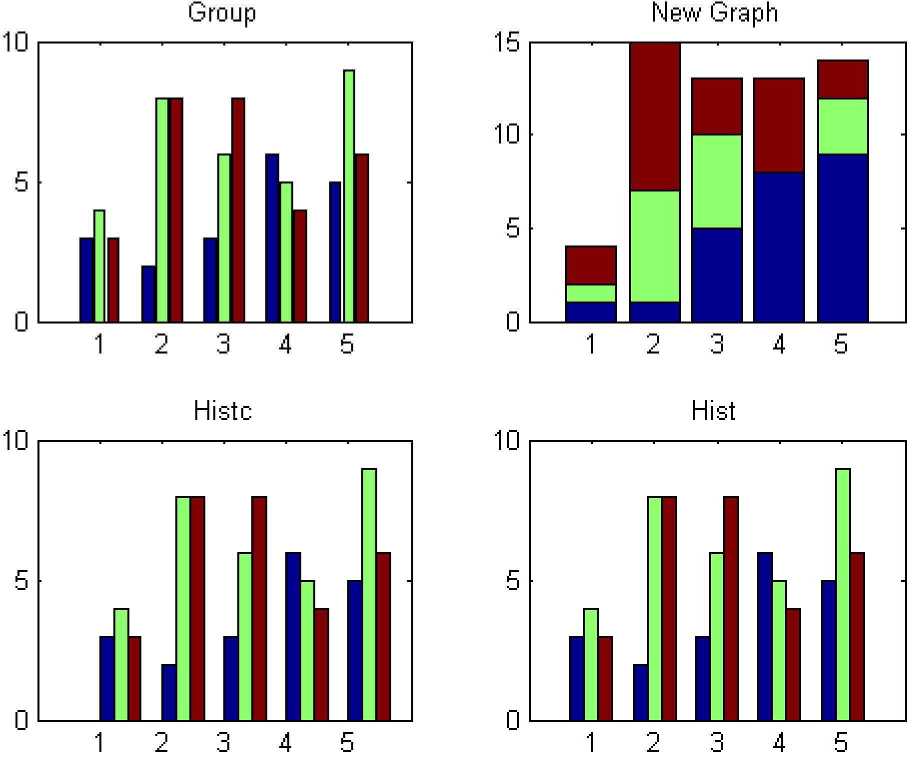




Specify Axes

Specify axes handle in the call to `bar`.

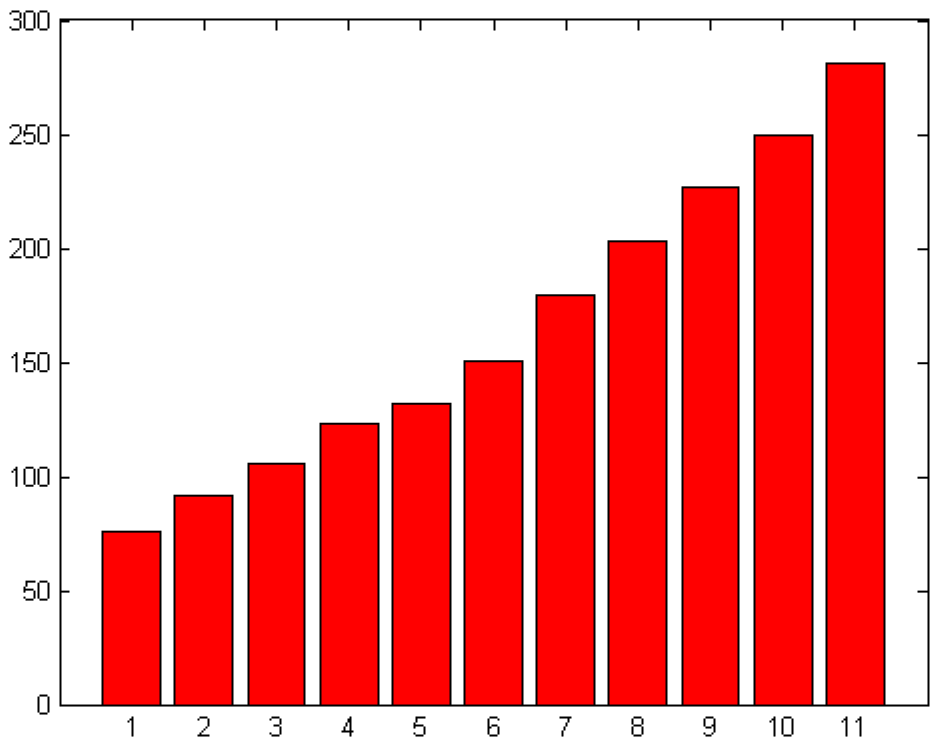
```
newY = round(rand(5,3)*10);
ah = subplot(2,2,2);
bar(ah,newY,'stacked'); title('New Graph')
```



Bar Color

Specify red color for the bar graph.

```
y = [75.995,91.972,105.711,123.203,131.669,...
     150.697,179.323,203.212,226.505,249.633,281.422];
figure; bar(y,'r');
```

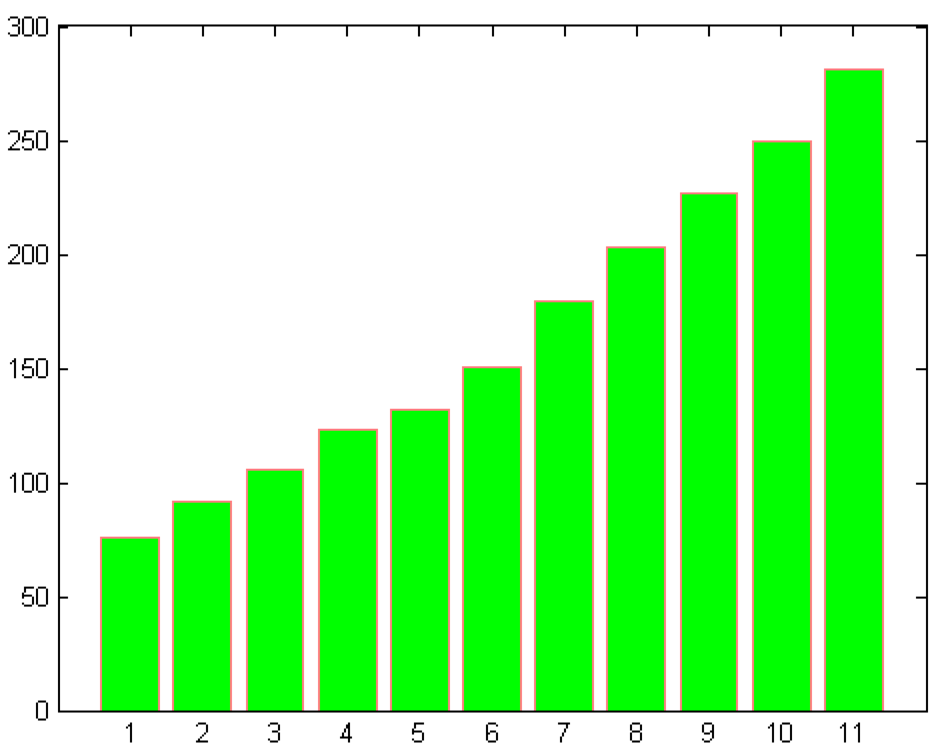


Bar Face and Edge Color

Specify green bars and use an RGB color specification for the bar edge color.

```
y = [75.995,91.972,105.711,123.203,131.669,...
     150.697,179.323,203.212,226.505,249.633,281.422];
```

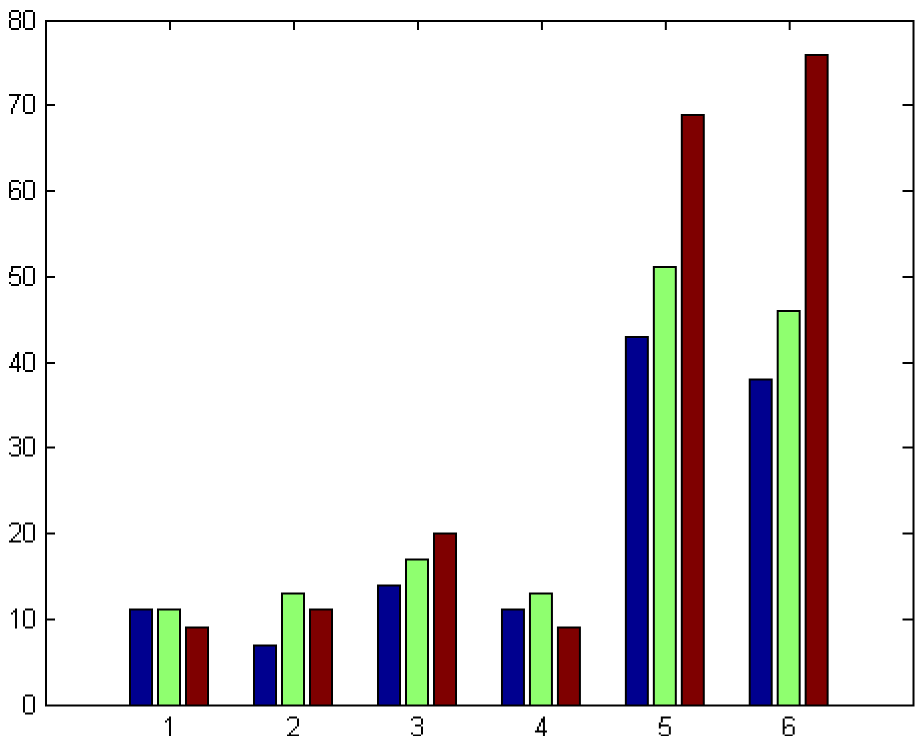
```
figure; bar(y,'g','EdgeColor',[1,0.5,0.5]);
```



Bar Graph of 2–D Array

Plot a 6–by-3 array as six groups with three bars each.

```
c = load('count.dat');
Y = c(1:6,:);
figure; bar(Y);
```



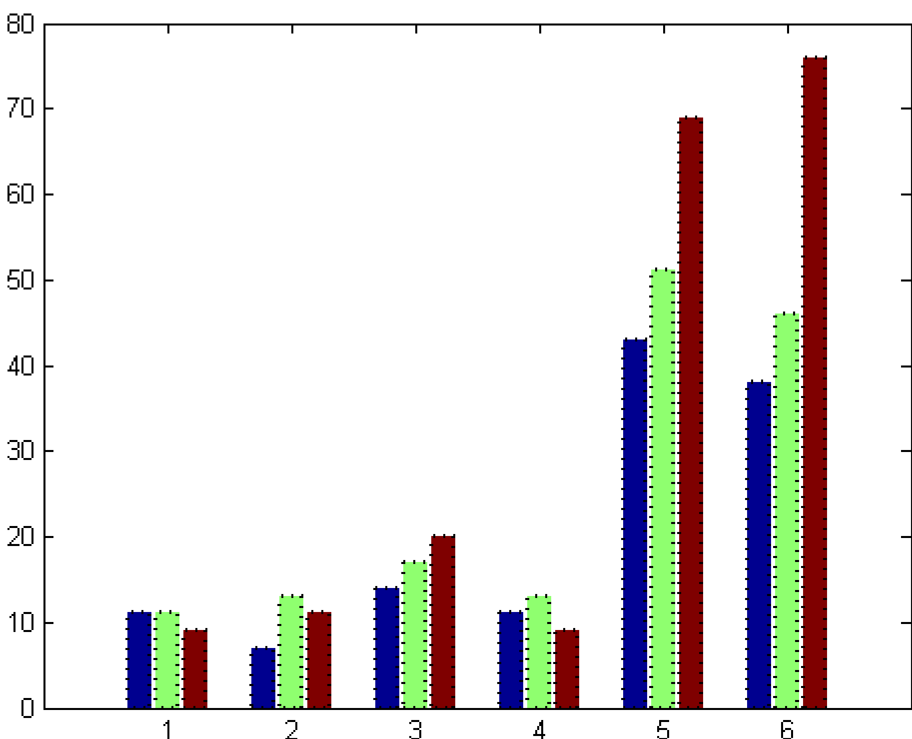
Emphasize Subset of Data

Set appearance of one column of data.

Set `LineWidth` and `EdgeColor` for the bars representing the second column in the data array `y`. Use the handles returned by `bar` and the `set` function to set the `barseries` properties.

```
c = load('count.dat');
Y = c(1:6,:);
hArray = bar(Y);
```

```
set(hArray(2),'LineWidth',2,'EdgeColor','red');
```

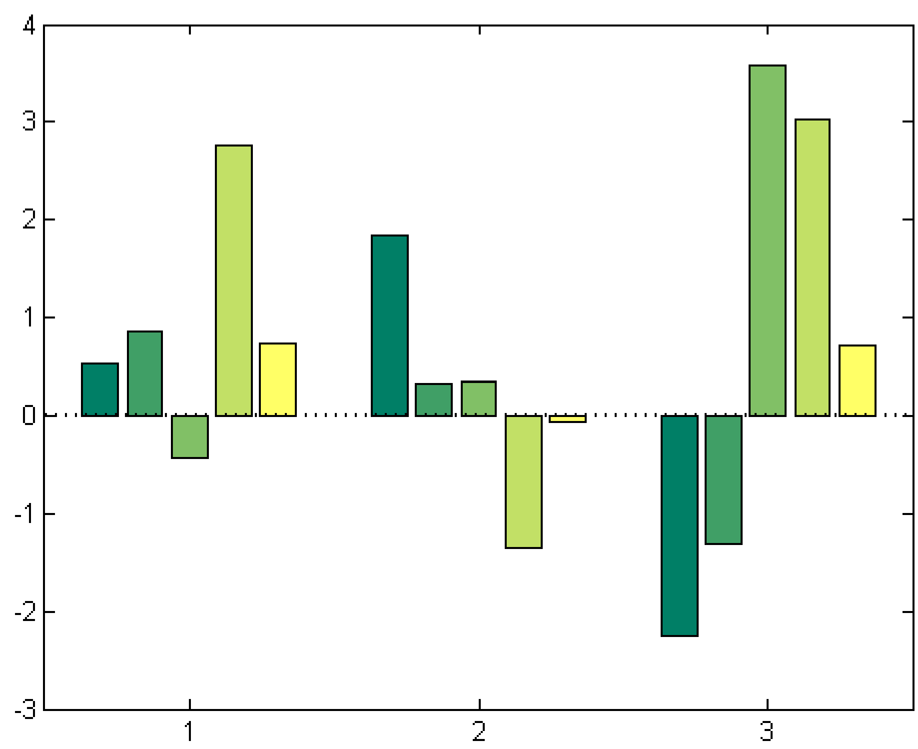


Baseline Modifications

Create a graph that displays three groups with five bars `barseries` objects.

All `barseries` objects in a graph share the same baseline. Use the `BaseLine` property of any `barseries` object to change the properties of the baseline. This example uses the first handle returned in `h`.

```
y = sin((-pi:pi).^2);
% Put the baseline at y = -0.2
h = bar(y,'BaseValue',-0.2);
% Get a handle to the baseline
hbl = get(h(1),'BaseLine');
% Change to a red dotted line
set(hbl,'Color','red','LineStyle',':')
```



See Also

`barh` | `bar3` | `bar3h` | `ColorSpec` | `stairs` | `hist`“Barseries Property Descriptions” on page 1-438

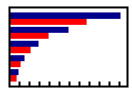
Related Examples

- “Bar and Area Graphs”

barh

Purpose

Plot bar graph horizontally



Syntax

```
barh(Y)
barh(X,Y)
barh(...,width)
barh(...,'style')
barh(...,'bar_color')
barh(...,'PropertyName',PropertyValue,...)
barh(axes_handle,...)
h = barh(...)
```

Description

A `barh` graph displays the values in a vector or matrix as horizontal bars.

`barh(Y)` draws one horizontal bar for each element in `Y`. If `Y` is a matrix, `barh` groups the bars produced by the elements in each row. The *x*-axis scale ranges from 1 up to `length(Y)` when `Y` is a vector, and `1` to `size(Y,1)`, which is the number of rows, when `Y` is a matrix. The default is to assign an appropriate progression of tick values according to the data. If you want the *x*-axis scale to end exactly at the last bar, set the axis limits as,

```
set(gca,'XLim',[1 length(Y)])
```

`barh(X,Y)` draws a bar for each element in `Y` at locations specified in `x`, where `X` is a vector defining the *x*-axis intervals for the vertical bars. The *x*-values can be nonmonotonic, but cannot contain duplicate values. If `Y` is a matrix, `barh` groups the elements of each row in `Y` at corresponding locations in `X`.

`barh(...,width)` sets the relative bar width and controls the separation of bars within a group. The default `width` is `0.8`, so if you do not specify `X`, the bars within a group have a slight separation. If

`width` is `1`, the bars within a group touch one another. The value of `width` must be a scalar.

`barh(...,'style')` specifies the style of the bars. `'style'` is `'grouped'` or `'stacked'`. Default mode of display is `'grouped'`.

- `'grouped'` displays *m* groups of *n* vertical bars, where *m* is the number of rows and *n* is the number of columns in `Y`. The group contains one bar per column in `Y`.
- `'stacked'` displays one bar for each row in `Y`. The bar height is the sum of the elements in the row. Each bar is multicolored, with colors corresponding to distinct elements and showing the relative contribution each row element makes to the total sum.
- `'histc'` displays the graph in histogram format, in which bars touch one another.
- `'hist'` also displays the graph in histogram format, but centers each bar over the *x*-ticks, rather than making bars span *x*-ticks as the `histc` option does.

Note When you use either the `hist` or `histc` option, you cannot also use parameter/value syntax. These two options create graphic objects that are patches rather than barseries.

`barh(...,'bar_color')` displays all bars using the color specified by the single-letter abbreviation `'r'`, `'g'`, `'b'`, `'c'`, `'m'`, `'y'`, `'k'`, or `'w'`.

`barh(...,'PropertyName',PropertyValue,...)` sets the named property or properties to the specified values. You cannot specify properties when `hist` or `histc` options are used. See the barseries property descriptions for information on what properties you can set.

`barh(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = barh(...)` returns a vector of handles to barseries graphics objects, one for each created. When `Y` is a matrix, `barh` creates one barseries graphics object per column in `Y`.

Barseries Objects

Creating a bar graph of an *m*-by-*n* matrix creates *m* groups of *n* barseries objects. Each barseries object contains the data for corresponding `x` values of each bar group (as indicated by the coloring of the bars).

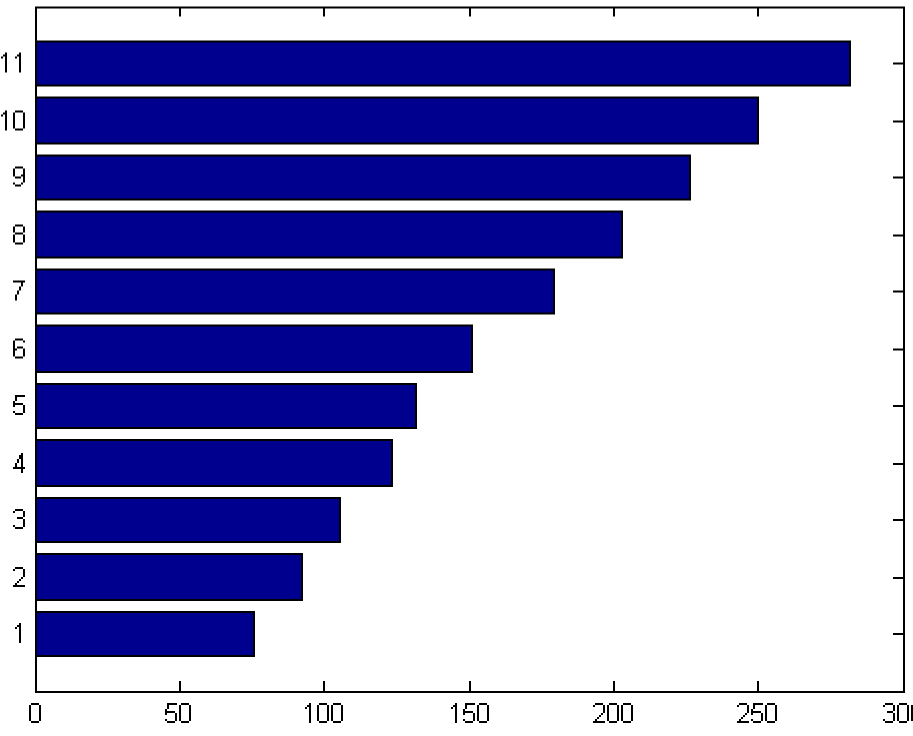
Note some barseries object properties set on an individual barseries object set the values for all barseries objects in the graph. See the barseries property descriptions for information on specific properties.

Examples

Single Series of Data

This example shows how to plot vector data using `barh` .

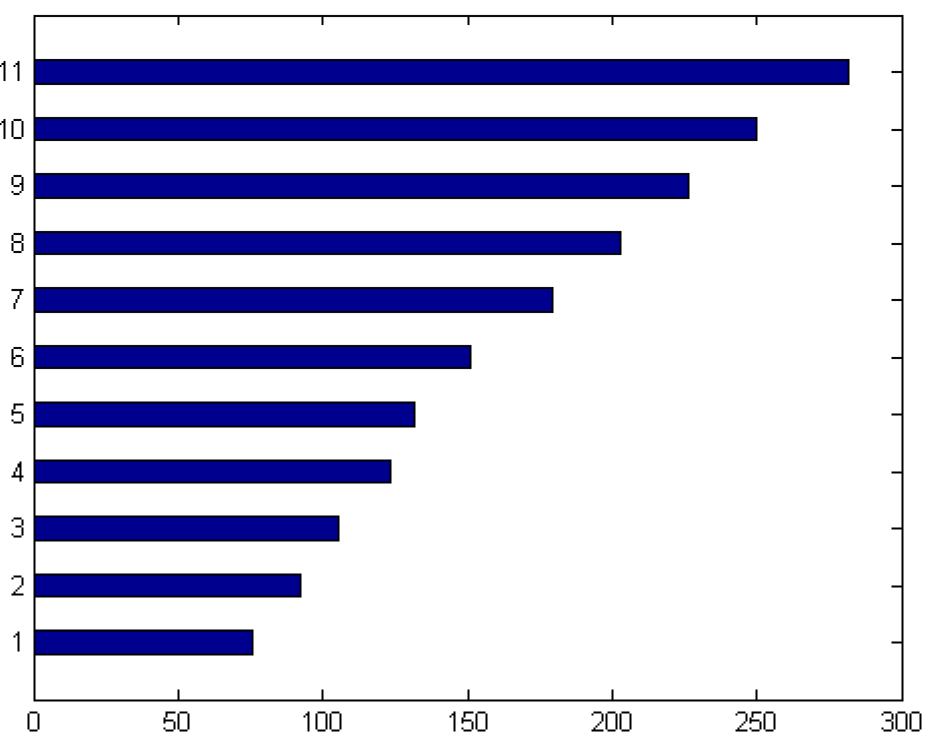
```
y = [75.995 91.972 105.711 123.203 131.669 ...
     150.697 179.323 203.212 226.505 249.633 281.422];
figure; barh(y);
```



Specifying width While Passing Single Variable.

This example shows how to specify width for each bar in the graph. You can specify one or both data inputs.

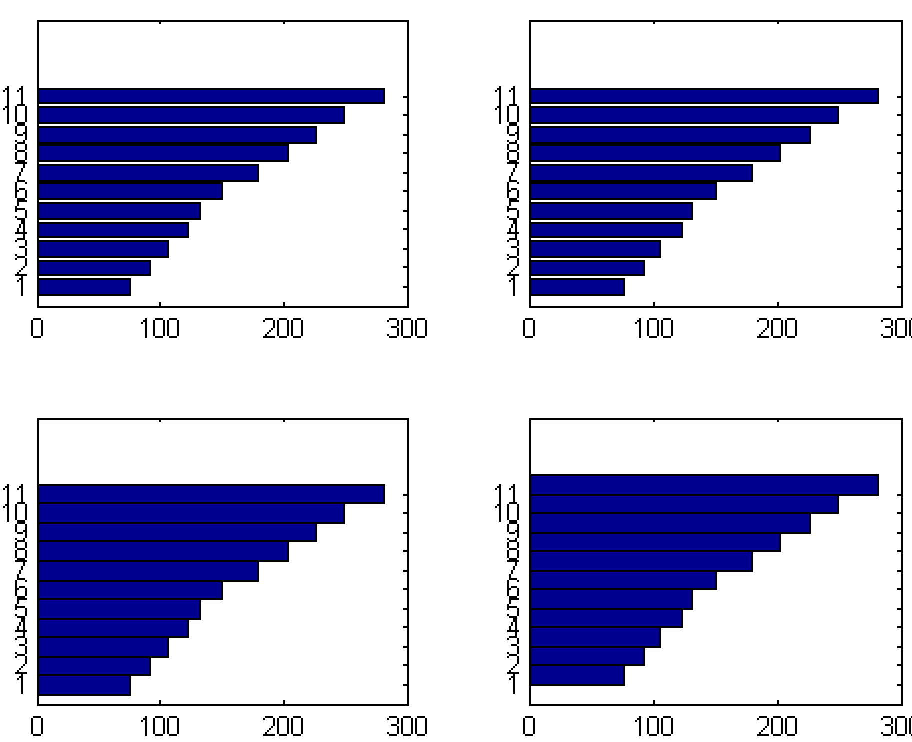
```
figure; barh(y,0.4);
```



Specifying style

This example shows you how to specify `Style` for `bar` graph.

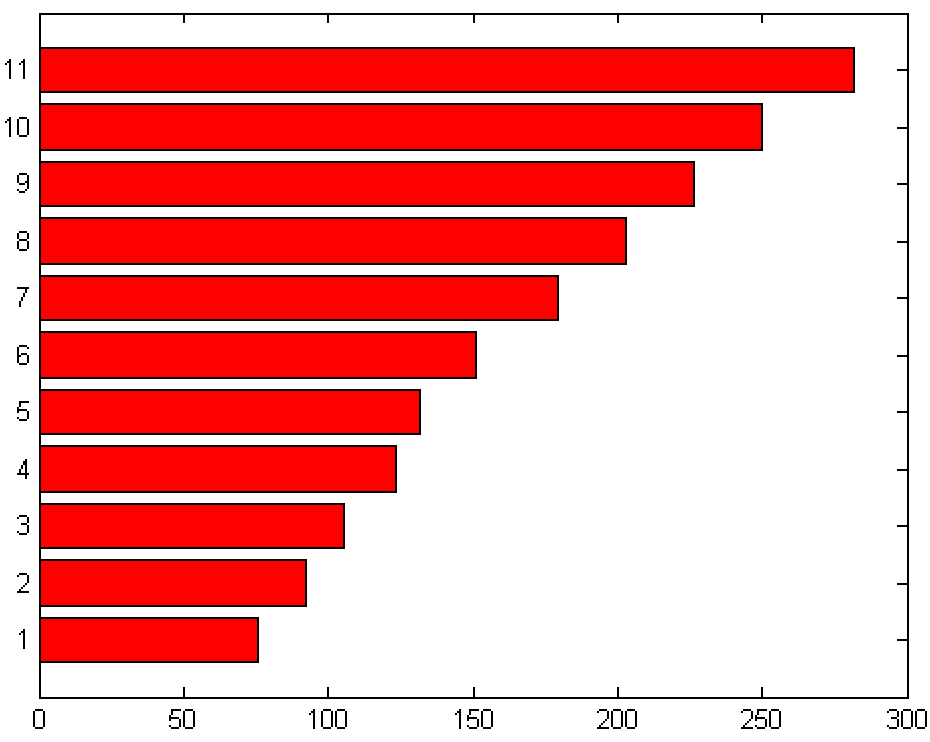
```
figure;
subplot(2,2,1); barh(y,'grouped');
subplot(2,2,2); barh(y,'stacked');
subplot(2,2,3); barh(y,'hist');
subplot(2,2,4); barh(y,'histc');
```



Specifying color

This example shows how to specify color for the graph.

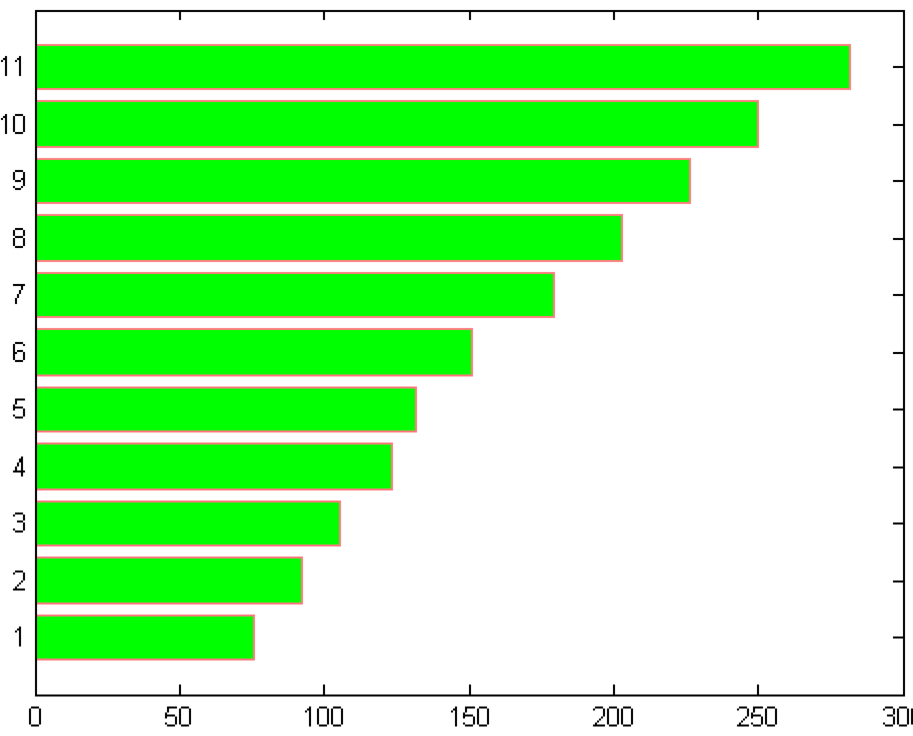
```
figure; barh(y,'r');
```



Specifying Name-Value Pairs

You can instead specify the rgb color instead of using the predefined ones. For example,

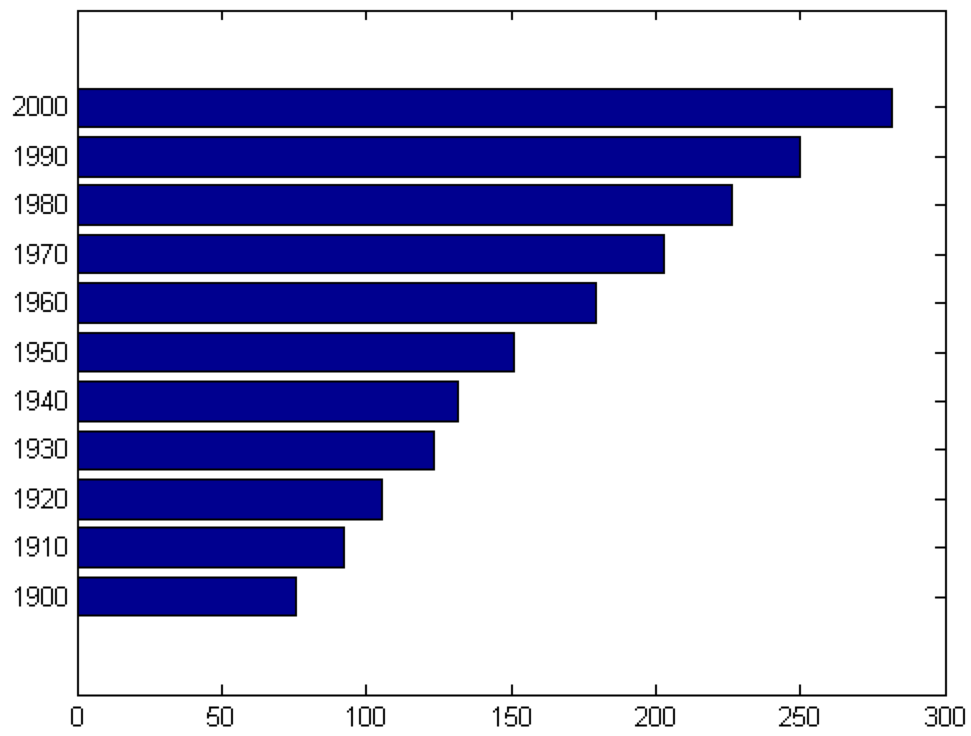
```
figure; barh(y,'g','EdgeColor',[1 0.5 0.5]);
```



Passing Two Variables

This example shows how to plot vector data with two inputs `x` and `y` using `barh` .

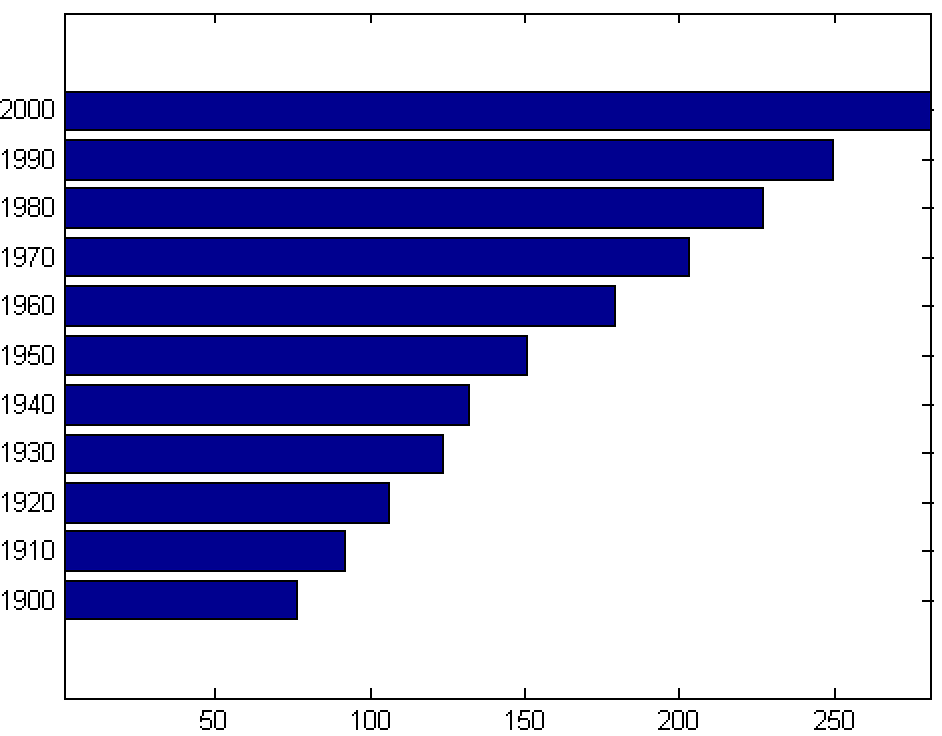
```
x = [1900:10:2000];
figure; barh(x,y);
```



Setting Axis Scale

The default is to assign an appropriate progression of tick values according to the data. If you want the *y*-axis scale to end exactly at the last bar, you can change the axes `YLim` property, which sets the limits of the *y*-axis. See `axes` for more information.

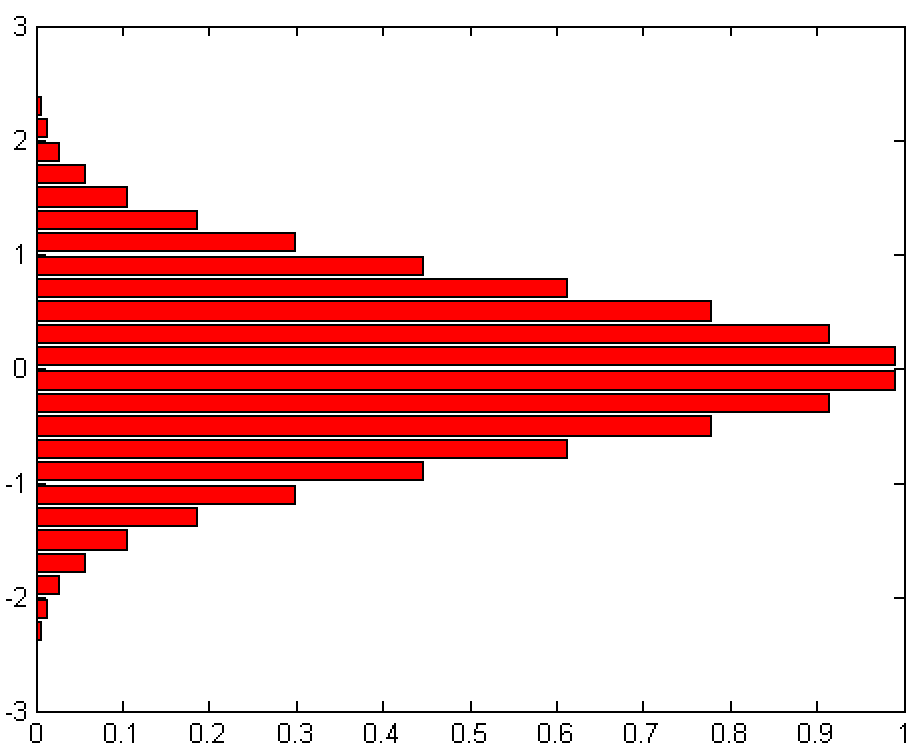
```
x = [1900:10:2000];
figure; barh(x,y);
set(gca,'YLim',[1 max(y)]);
```



Passing an Expression

You can also pass the expression with the `barh` function.

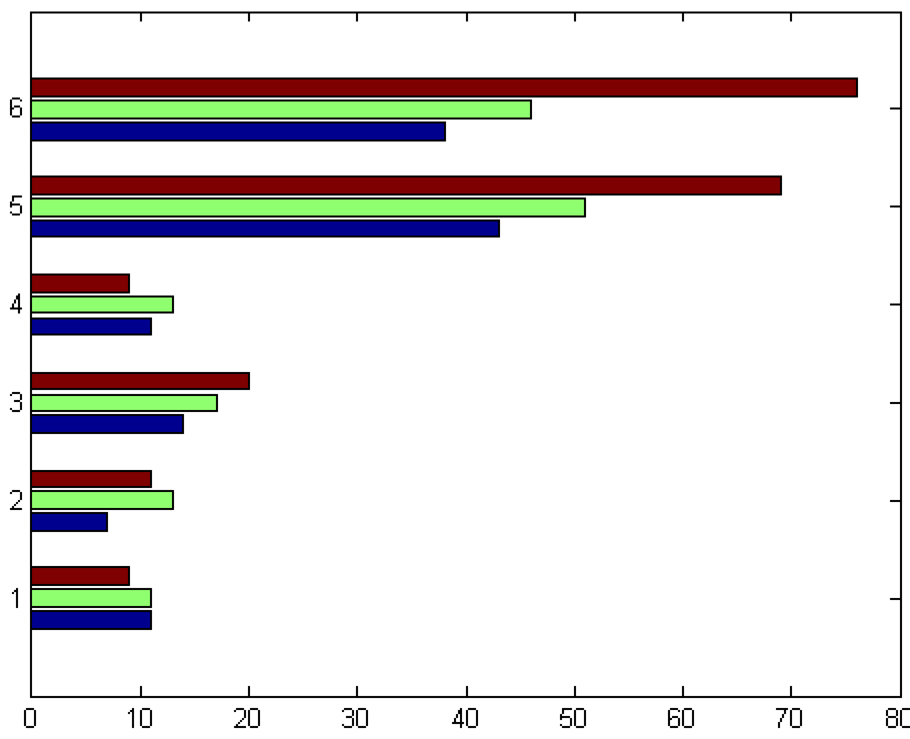
```
a = -2.9:0.2:2.9;
barh(a,exp(-a.*a),'r')
```



Plotting Matrix Data

This example shows how to plot matrix data using `barh`.

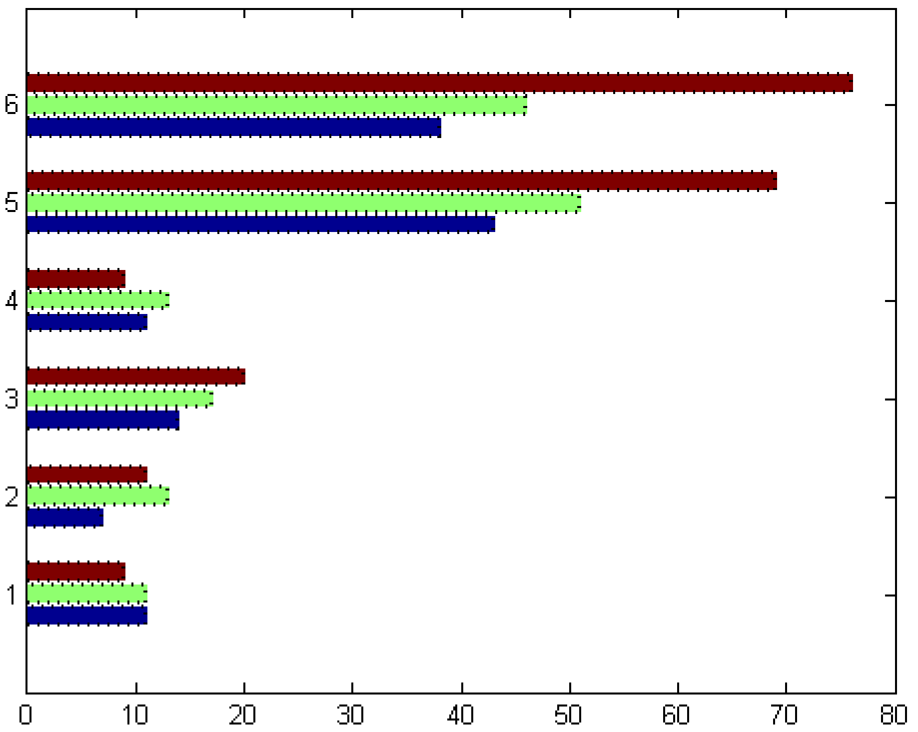
```
load count.dat;
yMat = count(1:6,:);
figure; barh(yMat);
```



Setting Properties with Multiobject Graphs

This example shows how to set `LineWidth` and `LineStyle` for three barseries objects using the handle returned by `barh`.

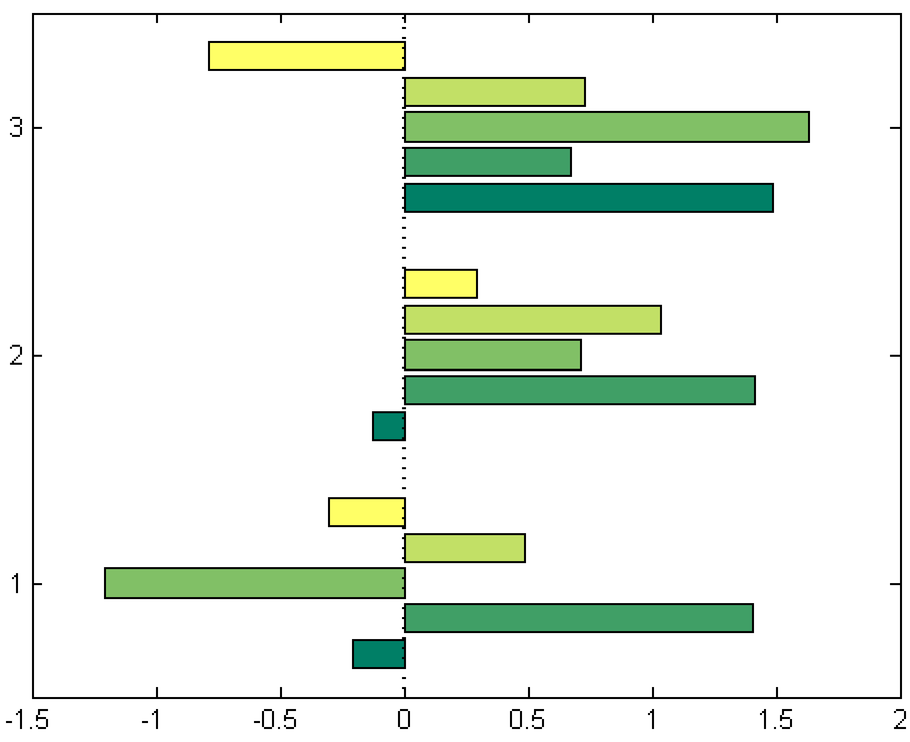
```
hMulti = barh(yMat);
set(hMulti,'LineWidth', 2, 'LineStyle',':');
```



Setting Properties with Multiobject Graphs

This example creates a graph that displays three groups of bars and contains five barseries objects. Since all barseries objects in a graph share the same baseline, you can set values using any barseries object's `BaseLine` property. This example uses the first handle returned in `h`.

```
Y = randn(3,5);
h = barh(Y);
set(get(h(1),'BaseLine'),'LineWidth',2,'LineStyle',':')
colormap summer % Change the color scheme
```



Specifying `colormap` assigns a specific color in the map spectrum to each handle object in the group resulting in one color for each object group.

See Also

`bar` | `bar3` | `bar3h` | `ColorSpec` | `stairs` | `hist` | `Barseries Properties`

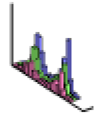
How To

- "Bar and Area Graphs"

bar3

Purpose

Plot 3-D bar graph



Syntax

```
bar3(Y)
bar3(x,Y)
bar3(...,width)
bar3(...,'style')
bar3(...,LineSpec)
bar3(axes_handle,...)
h = bar3(...)
```

Description

`bar3` draws a three-dimensional bar graph.

`bar3(Y)` draws a three-dimensional bar chart, where each element in `Y` corresponds to one bar. When `Y` is a vector, the *x*-axis scale ranges from `1` to `length(Y)`. When `Y` is a matrix, the *x*-axis scale ranges from `1` to `size(Y,1)` and the elements in each row are grouped together.

`bar3(x,Y)` draws a bar chart of the elements in `Y` at the locations specified in `x`, where `x` is a vector defining the *y*-axis intervals for vertical bars. The *x*-values can be nonmonotonic, but cannot contain duplicate values. If `Y` is a matrix, `bar3` clusters elements from the same row in `Y` at locations corresponding to an element in `x`. Values of elements in each row are grouped together.

`bar3(...,width)` sets the width of the bars and controls the separation of bars within a group. The default `width` is `0.8`, so if you do not specify `x`, bars within a group have a slight separation. If `width` is `1`, the bars within a group touch one another.

`bar3(...,'style')` specifies the style of the bars. `'style'` is `'detached'`, `'grouped'`, or `'stacked'`. Default mode of display is `'detached'`.

- `'detached'` displays the elements of each row in `Y` as separate blocks behind one another in the *x* direction.
- `'grouped'` displays *n* groups of *m* vertical bars, where *n* is the number of rows and *m* is the number of columns in `Y`. The group contains one bar per column in `Y`.
- `'stacked'` displays one bar for each row in `Y`. The bar height is the sum of the elements in the row. Each bar is multicolored, with colors corresponding to distinct elements and showing the relative contribution each row element makes to the total sum.

`bar3(...,LineSpec)` displays all bars using the color specified by `LineSpec`.

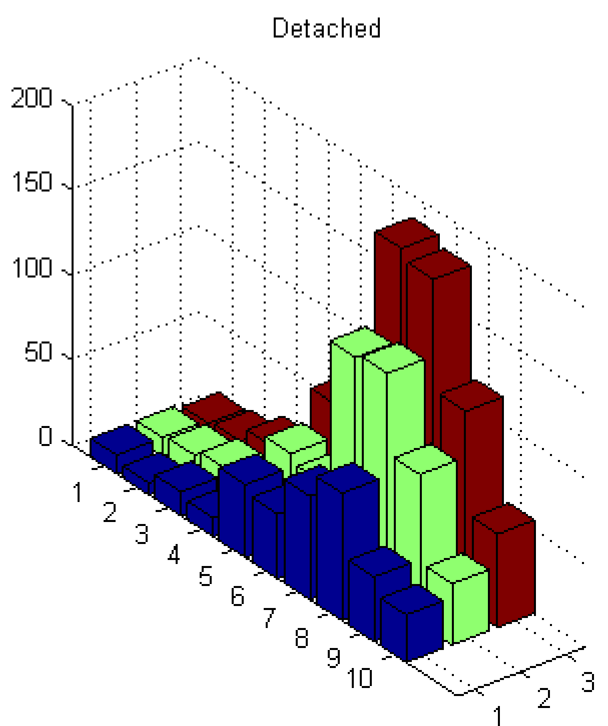
`bar3(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

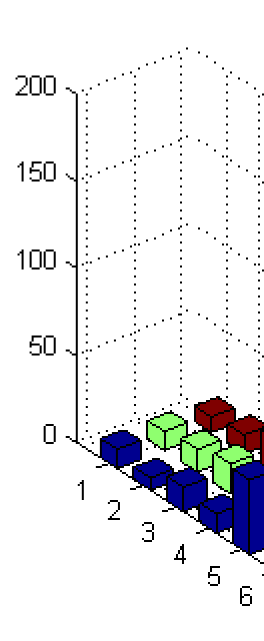
`h = bar3(...)` returns a vector of handles to patch graphics objects, one for each created. `bar3` creates one patch object per column in `Y`. When `Y` is a matrix, `bar3` creates one patch graphics object per column in `Y`.

Examples

Compare two 3-D bar graphs where style is specified as `'detached'` and width is varied. This type of graph helps in visualizing trend of each row in the input data.

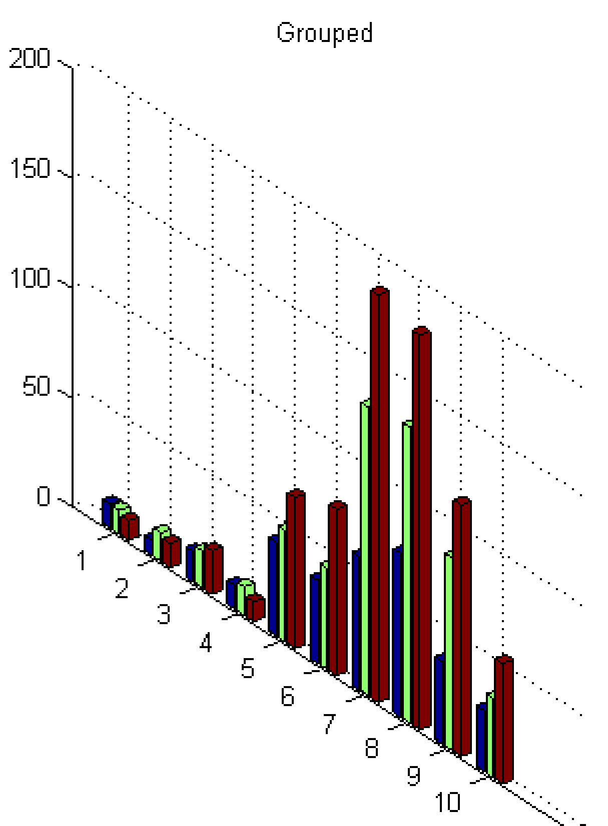
```
load count.dat;
y = count(1:10,:); % Loading the dataset creates a variable 'count'
figure; subplot(1,2,1);
bar3(y,'detached');
title('Detached');
subplot(1,2,2);
bar3(y,0.5,'detached');
title('Width = 0.5');
```

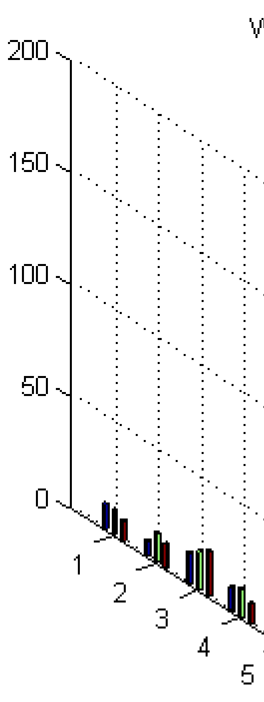

Detached
200
150
100
50
0
1
2
3
4
5
6
7
8
9
10
1
2
3


200
150
100
50
0
1
2
3
4
5
6

Create a graph where width is varied and style is `'grouped'`. The `'grouped'` graph gives a better visualization for variation of data at each point.

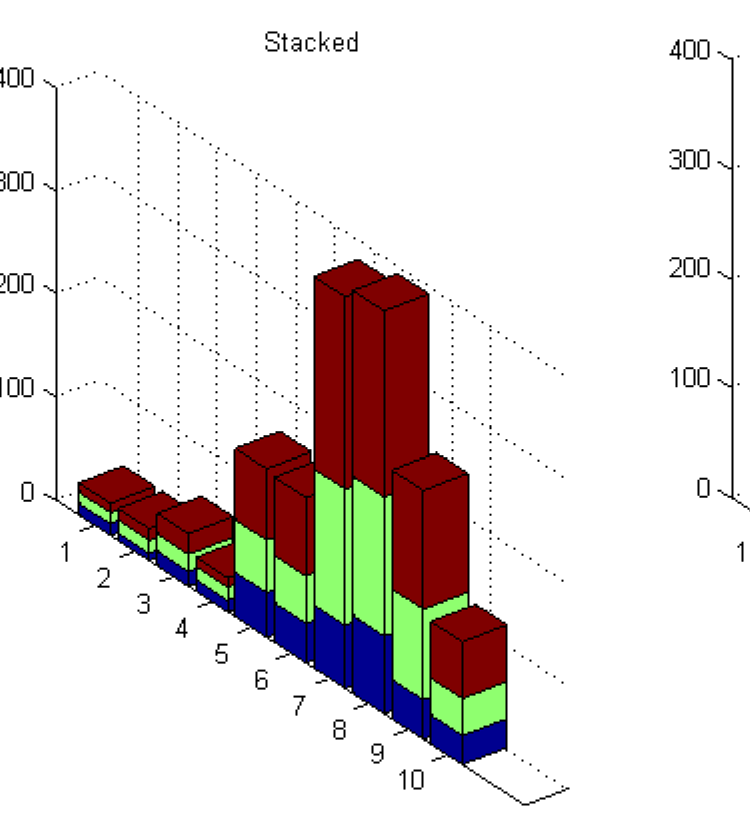
```
figure; subplot(1,2,1);
bar3(y,'grouped');
title('Grouped');
subplot(1,2,2);
bar3(y,0.25,'grouped');
title('Width = 0.25');
```


Grouped
200
150
100
50
0
1
2
3
4
5
6
7
8
9
10


200
150
100
50
0
1
2
3
4
5

Create a 3-D bar graph with the style option `'stacked'`. A `'stacked'` graph visualizes the cumulative effect of the data at each data point.

```
figure; subplot(1,2,1);
bar3(y,'stacked');
title('Stacked');
subplot(1,2,2);
bar3(y,0.25,'stacked');
title('Width = 0.25');
```


Stacked
400
300
200
100
0
1
2
3
4
5
6
7
8
9
10
400
300
200
100
0
1
2
3
4
5

See Also

`bar` | `barh` | `bar3h` | `LineSpec`

How To

- "Bar and Area Graphs"

bar3h

Purpose

Plot horizontal 3-D bar graph

Syntax

```
bar3h(Y)
bar3h(x,Y)
bar3h(...,width)
bar3h(...,'style')
bar3h(...,LineSpec)
bar3h(axes_handle,...)
h = bar3h(...)
```

Description

`bar3h` draws three-dimensional horizontal bar charts.

`bar3h(Y)` draws a three-dimensional bar chart, where each element in `Y` corresponds to one bar. When `Y` is a vector, the *y*-axis scale ranges from `1` to `length(Y)`. When `Y` is a matrix, the *y*-axis scale ranges from `1` to `size(Y,1)` and the elements in each row are grouped together.

`bar3h(x,Y)` draws a bar chart of the elements in `Y` at the locations specified in `x`, where `x` is a vector defining the *y*-axis intervals for horizontal bars. The *x*-values can be nonmonotonic, but cannot contain duplicate values. If `Y` is a matrix, `bar3` clusters elements from the same row in `Y` at locations corresponding to an element in `x`. Values of elements in each row are grouped together.

`bar3h(...,width)` sets the width of the bars and controls the separation of bars within a group. The default `width` is `0.8`, so if you do not specify `x`, bars within a group have a slight separation. If `width` is `1`, the bars within a group touch one another.

`bar3h(...,'style')` specifies the style of the bars. `'style'` is `'detached'`, `'grouped'`, or `'stacked'`. Default mode of display is `'detached'`.

- `'detached'` displays the elements of each row in `Y` as separate blocks behind one another in the *y* direction.

- `'grouped'` displays *n* groups of *m* vertical bars, where *n* is the number of rows and *m* is the number of columns in `Y`. The group contains one bar per column in `Y`.
- `'stacked'` displays one bar for each row in `Y`. The bar length is the sum of the elements in the row. Each bar is multicolored, with colors corresponding to distinct elements and showing the relative contribution each row element makes to the total sum.

`bar3h(...,LineSpec)` displays all bars using the color specified by `LineSpec`.

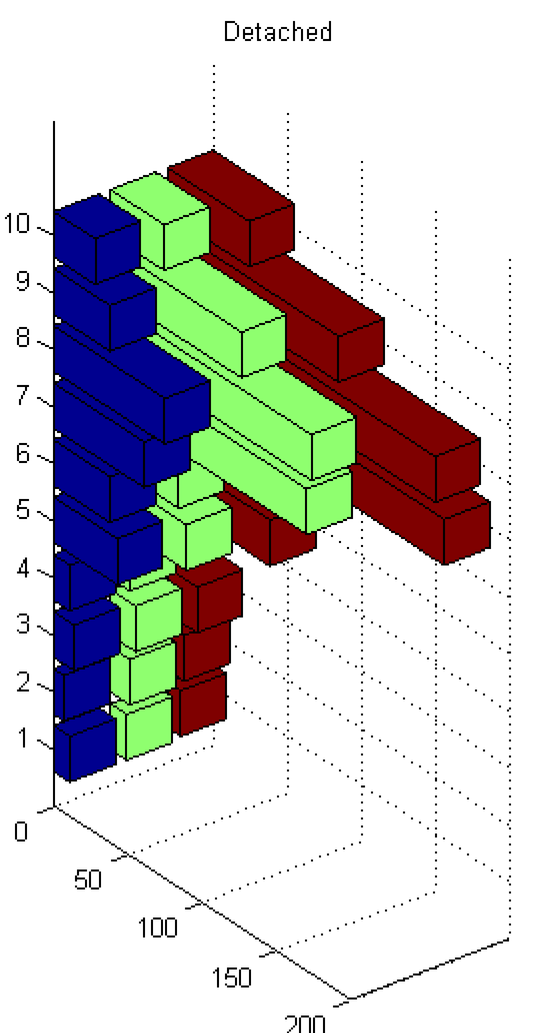
`bar3h(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

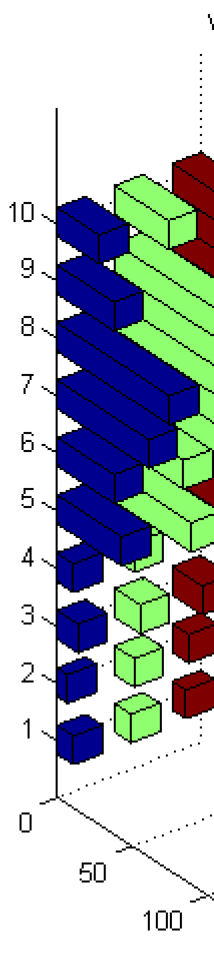
`h = bar3h(...)` returns a vector of handles to patch graphics objects, one for each created. `bar3` creates one patch object per column in `Y`. When `Y` is a matrix, `bar3h` creates one patch graphics object per column in `Y`.

Examples

Compare two 3-D bar graphs where style is specified as `'detached'` and width is varied. This type of graph helps in visualizing trend of each row in the input data.

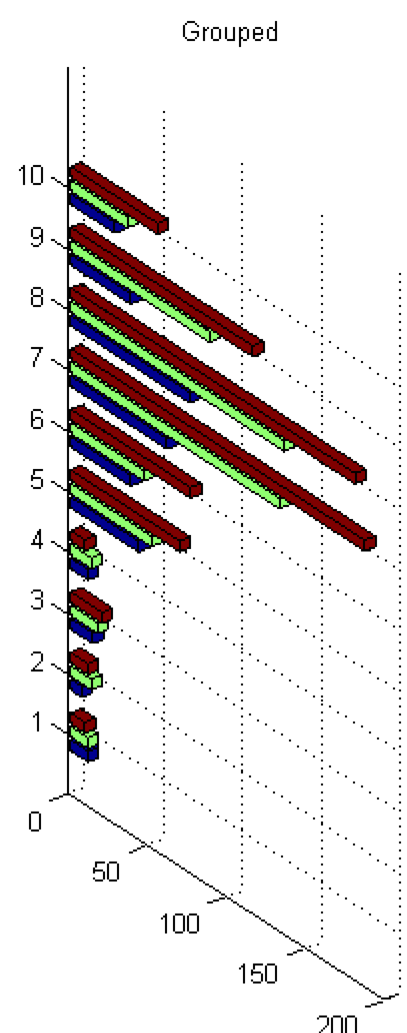
```
load count.dat;
y = count(1:10,:); % Loading the dataset creates a variable 'count'
figure; subplot(1,2,1);
bar3h(y,'detached');
title('Detached');
subplot(1,2,2);
bar3h(y,0.5,'detached');
title('Width = 0.5');
```

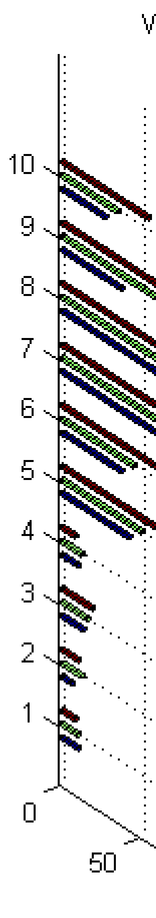

Detached
10
9
8
7
6
5
4
3
2
1
0
50
100
150
200


10
9
8
7
6
5
4
3
2
1
0
50
100

Create a graph where width is varied and style is `'grouped'`. The `'grouped'` graph gives a better visualization for variation of data at each point.

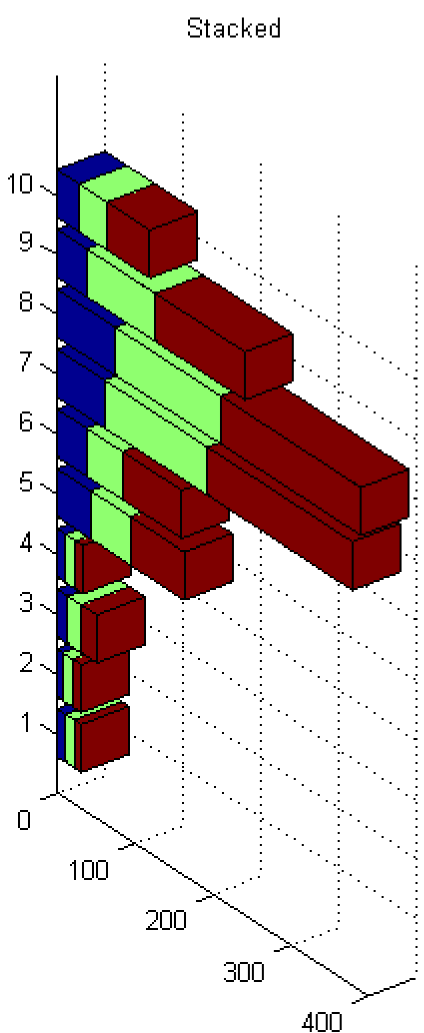
```
figure; subplot(1,2,1);
bar3h(y,'grouped');
title('Grouped');
subplot(1,2,2);
bar3h(y,0.25,'grouped');
title('Width = 0.25');
```

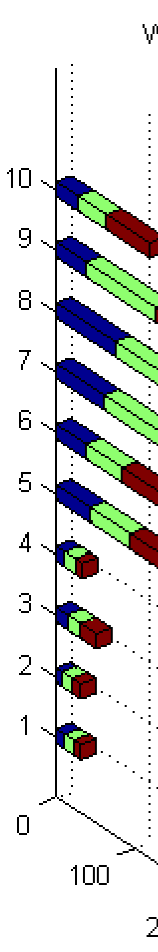

Grouped
10
9
8
7
6
5
4
3
2
1
0
50
100
150
200


10
9
8
7
6
5
4
3
2
1
0
50

Create a 3-D horizontal bar graph with the style option `'stacked'`. A `'stacked'` graph visualizes the cumulative effect of the data at each data point.

```
figure; subplot(1,2,1);
bar3h(y,'stacked');
title('Stacked');
subplot(1,2,2);
bar3h(y,0.25,'stacked');
title('Width = 0.25');
```


Stacked
10
9
8
7
6
5
4
3
2
1
0
100
200
300
400


10
9
8
7
6
5
4
3
2
1
0
100

See Also

`bar` | `barh` | `bar3` | `LineSpec` | `patch`

How To

- “Bar and Area Graphs”

Barseries Properties

Purpose

Define barseries properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands or the Property Editor (`propertyeditor`).

Note that you cannot define default properties for barseries objects.

See "Plot Objects" for more information on barseries objects.

Barseries Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`Annotation`

`hg.Annotation` object (read-only)

Control the display of barseries objects in legends. Specifies whether this barseries object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the barseries object is displayed in a figure legend.

IconDisplayStyle Value	Purpose
`on`	Include the barseries object in a legend as one entry, but not its children objects
`off`	Do not include the barseries or its children in a legend (default)
`children`	Include only the children of the barseries as separate entries in the legend

Setting the IconDisplayStyle Property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the IconDisplayStyle Property

See "Controlling Legends" for more information and examples.

`BarLayout`

`{grouped} | stacked`

Specify grouped or stacked bars.

- `grouped` — Display *m* groups of *n* vertical bars, where *m* is the number of rows and *n* is the number of columns in the input argument `Y`. The group contains one bar per column in `Y`.
- `stacked` — Display one bar for each row in the input argument `Y`. The bar height is the sum of the elements in the row. Each bar is multicolored, with colors corresponding to distinct elements and showing the relative contribution each row element makes to the total sum.

`BarWidth`

scalar in range `[0 1]`

Width of individual bars. `BarWidth` specifies the relative bar width and controls the separation of bars within a group. The default is `0.8`, so if you do not specify `x`, the bars within a group have a slight separation. If `BarWidth` is `1`, the bars within a group touch one another.

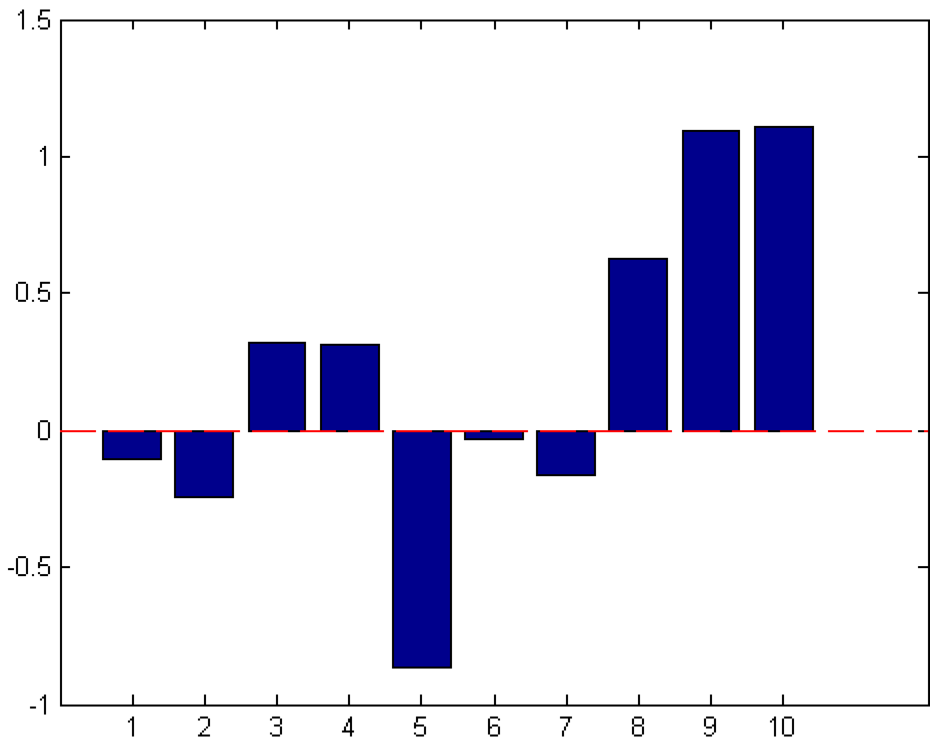
`BaseLine`

handle

Handle of the baseline line object. Handle of the line object used as the baseline. You can set the properties of this line using its handle.

Create a bar graph, obtain the handle of the baseline from the barseries object, and then set line properties that make the baseline a dashed, red line:

```
bar_handle = bar(randn(10,1));
baseline_handle = get(bar_handle,'BaseLine');
set(baseline_handle,'LineStyle','--','Color','red')
```



`BaseValue`

double: y-axis value

Baseline location. You can specify the value along the *y*-axis (vertical bars) or *x*-axis (horizontal bars) at which the MATLAB software draws the baseline. The default is `0`.

`BeingDeleted`

`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`
string | function handle

Button press callback function. Executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be:

- A string that is a valid MATLAB expression
- The name of a MATLAB file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`Children`
array of graphics object handles

Children of the barseries object. The handle of a patch object that is the child of this object (whether visible or not).

If a child object's `HandleVisibility` property is `callback` or `off`, its handle does not show up in this object's `Children` property. If you want the handle in the `Children` property, set the root `ShowHiddenHandles` property to `on`. For example:

```
set(0,'ShowHiddenHandles','on')
```

`Clipping`
`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs appear outside the axes plot box. This occurs if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`CreateFcn`
function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback function executed during object creation. Defines a callback that executes when MATLAB creates an object. The default is an empty array.

You must specify the callback during the creation of the object. For example:

```
graphicfcn(y,'CreateFcn',@CallbackFcn)
```

where *`@CallbackFcn`* is a function handle that references the callback function and *`graphicfcn`* is the plotting function which creates this object.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DeleteFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback executed during object deletion. Executes when this object is deleted (for example, this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values. The default is an empty array.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`

string

String used by legend. The `legend` function uses the `DisplayName` property to label the barseries object in the legend. The default is an empty string.

- If you specify string arguments with the `legend` function, MATLAB set `DisplayName` to the corresponding string and uses that string for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, MATLAB set `DisplayName` to the edited string.

- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add a legend programmatically that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more information and examples.

`EdgeColor`
`{[0 0 0]} | none | ColorSpec`

Color of line separating filled areas. You can set the color of the edges of filled areas to a three-element RGB vector or one of the MATLAB predefined names, including the string `none`. The default value is `[0 0 0]` (`black`). See the `ColorSpec` reference page for more information on specifying color.

`EraseMode`
`{normal} | none | xor | background`

Erase mode. Controls the technique MATLAB uses to draw and erase objects. Use alternative erase modes for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the object when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it because MATLAB stores no information about its former location.
- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However,

the object's color depends on the color of whatever is beneath it on the display.

- `background` — Erase the object by redrawing it in the axes background `Color`, or the figure background `Color` if the axes `Color` property is `none`. This damages objects that are behind the erased object, but properly colors the erased object.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors (for example, performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`FaceColor`
`{flat} | none | ColorSpec`

Color of filled areas.

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for all filled areas.
- `none` — Do not draw faces. Note that MATLAB draws `EdgeColor` independently of `FaceColor`.

- `flat` — The object uses the figure `colormap` to determine the color of the filled areas.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible.
- `callback` — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties. See also `findall`.

Handle Validity

Hidden handles are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`

`{on} | off`

Selectable by mouse click. Determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the objects that compose the bar graph. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`HitTestArea`

`on | {off}`

Select barseries object on bars or area of extent. This property enables you to select barseries objects in two ways:

- Select by clicking bars (default).

- Select by clicking anywhere in the extent of the bar graph.

When `HitTestArea` is `off`, you must click the bars to select the barseries object. When `HitTestArea` is `on`, you can select the barseries object by clicking anywhere within the extent of the bar graph (that is, anywhere within a rectangle that encloses all the bars).

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the `Interruptible` property affects only the callbacks for the`ButtonDownFcn` property. A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both the callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see “Control Callback Execution and Interruption”.

`LineStyle`

`{-} | -- | :  | -.  | none`

Line style of barseries object.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line
'none'	No line

`LineWidth`
size in points

Width of linear objects and edges of filled areas. Specify in points. 1 point = $^1/_{72}$ inch. The default is `0.5` points.

`Parent`
handle of parent axes, hggroup, or hgtransform

Parent of object. Handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Object selection state. When you set this property to `on`, MATLAB displays selection handles at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
`{on} | off`

Object highlighted when selected.

- `on` — MATLAB indicates the selected state by drawing four edge handles and four corner handles.
- `off` — MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`ShowBaseLine`
`{on} | off`

Turn baseline display on or off. This property determines whether bar plots display a baseline from which the bars are drawn. By default, the baseline is displayed.

`Tag`

string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

For example, create a barseries object and set the `Tag` property:

```
t = bar(Y,'Tag','bar1')
```

To access the barseries object, use `findobj` to find the barseries object's handle. The following statement changes the `FaceColor` property of the object whose `Tag` is `bar1`.

```
set(findobj('Tag','bar1'),'FaceColor','red')
```

`Type`

string (read-only)

Type of graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For barseries objects, `Type` is '`hggroup`'.

The following statement finds all the hggroup objects in the current axes.

```
t = findobj(gca,'Type','hggroup');
```

`UIContextMenu`

handle of uicontextmenu object

Associate context menu with object. Handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object. The default value is an empty array.

`UserData`
array

User-specified data. Data you want to associate with this object (including cell arrays and structures). The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`
`{on} | off`

Visibility of object and its children.

- `on` — Object and all children of the object are visible unless the child object's `Visible` property is `off`.
- `off` — Object not displayed. However, the object still exists and you can set and query its properties.

`XData`
array

Location of bars. The *x*-axis intervals for the vertical bars or *y*-axis intervals for horizontal bars (as specified by the `X` input argument). If `YData` is a vector, `XData` must be the same size. If `YData` is a matrix, the length of `XData` must be equal to the number of rows in `YData`.

`XDataMode`
`{auto} | manual`

Use automatic or user-specified x-axis values. If you specify `XData` (by setting the `XData` property or specifying the `X` input

argument), MATLAB sets this property to `manual` and uses the specified values to label the *x*-axis.

If you set `XDataMode` to `auto` after specifying `XData`, MATLAB resets the *x*-axis ticks to `1:size(YData,1)` or to the column indices of the `ZData`, overwriting any previous values for `XData`.

`XDataSource`

MATLAB variable, as a string

Link `XData` *to MATLAB variable.* Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `XData`. The default value is an empty array.

```
set(h,'XDataSource','xdatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `XDataSource` does not change the object's `XData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`YData`

scalar | vector | matrix

Bar plot data. `YData` contains the data plotted as bars (the `Y` input argument). Each value in `YData` is represented by a bar in the bar graph. If `XYData` is a matrix, the `bar` function creates a "group"

or a "stack" of bars for each column in the matrix. See the `bar` reference page for examples of grouped and stacked bar graphs.

The input argument `Y` in the `bar` function calling syntax assigns values to `YData`.

`YDataSource`

MATLAB variable, as a string

Link `YData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `YData`. The default value is an empty array.

```
set(h,'YDataSource','Ydatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `YDataSource` does not change the object's `YData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

How To

- "Bar and Area Graphs"
- "Plot Objects"
- "Setting Default Property Values"
- "The Property Editor"

TriRep.baryToCart

Purpose

(Will be removed) Convert point coordinates from barycentric to Cartesian

---

Note `baryToCart(TriRep)` will be removed in a future release. Use `barycentricToCartesian(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
XC = baryToCart(TR, SI, B)
```

Description

`XC = baryToCart(TR, SI, B)` returns the Cartesian coordinates `XC` of each point in `B` that represents the barycentric coordinates with respect to its associated simplex `SI`.

Input Arguments

`TR`	Triangulation representation.
`SI`	Column vector of simplex indices that index into the triangulation matrix `TR.Triangulation`
`B`	`B` is a matrix that represents the barycentric coordinates of the points to convert with respect to the simplices `SI`. `B` is of size `m`-by-`k`, where `m` = `length(SI)`, the number of points to convert, and `k` is the number of vertices per simplex.

Output Arguments

`XC`	Matrix of cartesian coordinates of the converted points. `XC` is of size `m`-by-`n`, where `n` is the dimension of the space where the triangulation resides. That is, the Cartesian coordinates of the point `B(j)` with respect to simplex `SI(j)` is `XC(j)`.

Definitions

A simplex is a triangle/tetrahedron or higher-dimensional equivalent.

Examples

Compute the Delaunay triangulation of a set of points.

```
x = [0 4 8 12 0 4 8 12]';
y = [0 0 0 0 8 8 8 8]';
dt = DelaunayTri(x,y)
```

Compute the barycentric coordinates of the incenters.

```
cc = incenters(dt);
tri = dt(:,:);
```

Plot the original triangulation and reference points.

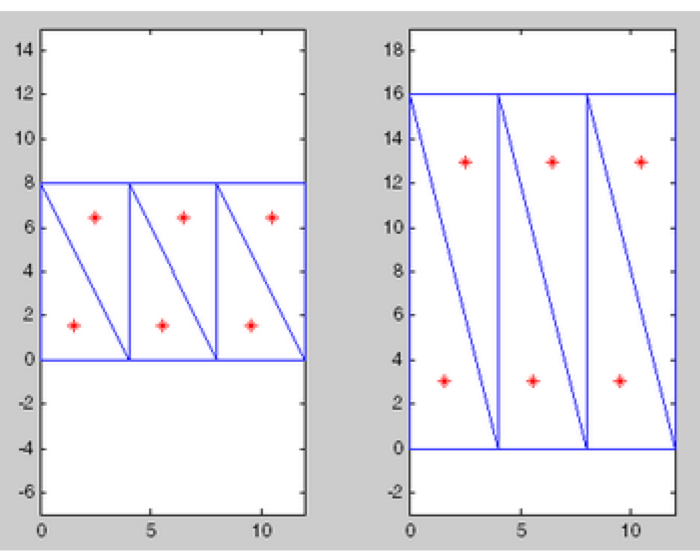
```
figure
subplot(1,2,1);
triplot(dt); hold on;
plot(cc(:,1), cc(:,2), '*r'); hold off;
axis equal;
```

Stretch the triangulation and compute the mapped locations of the incenters on the deformed triangulation.

```
b = cartToBary(dt,[1:length(tri)]',cc);
y = [0 0 0 0 16 16 16 16]';
tr = TriRep(tri,x,y)
xc = baryToCart(tr, [1:length(tri)]', b);
```

Plot the deformed triangulation and mapped locations of the reference points.

```
subplot(1,2,2);
triplot(tr); hold on;
plot(xc(:,1), xc(:,2), '*r'); hold off;
axis equal;
```



See Also

`cartesianToBarycentric` | `pointLocation` | `delaunayTriangulation` | `triangulation`

Purpose

Convert base N number string to decimal number

Syntax

```
d = base2dec('strn', base)
```

Description

`d = base2dec('strn', base)` converts the string number *`strn`* of the specified `base` into its decimal (base 10) equivalent. `base` must be an integer between 2 and 36. If `'strn'` is a character array, each row is interpreted as a string in the specified `base`.

Examples

The expression `base2dec('212',3)` converts 212_3 to decimal, returning 23.

See Also

`dec2base`

beep

Purpose

Produce beep sound

Syntax

```
beep
beep on
beep off
s = beep
```

Description

`beep` produces your computer's default beep sound.

`beep on` turns the beep on.

`beep off` turns the beep off.

`s = beep` returns the current beep mode (`on` or `off`).

Purpose

Initiate asynchronous .NET delegate call

Syntax

```
result = BeginInvoke(arg1,...,argN,callback,object)
```

Description

`result = BeginInvoke(arg1,...,argN,callback,object)` initiates asynchronous call to a .NET delegate. You must call `EndInvoke` to complete the asynchronous call.

Input Arguments

arg1,...,argN

Input arguments for delegate. The type and number of arguments must agree with the delegate signature.

callback

.NET System.AsyncCallback delegate, or `[]` null value.

object

User-defined object, or `[]` null value.

Output Arguments

result

.NET System.IAsyncResult object. Used to monitor the progress of the asynchronous call. Input argument to `EndInvoke`.

See Also

`EndInvoke`

How To

- “Calling .NET Methods Asynchronously”

Related Links

- MSDN® Calling Synchronous Methods Asynchronously

bench

Purpose

MATLAB benchmark

Syntax

```
bench
bench(N)
bench(0)
t = bench(N)
```

Description

`bench` times six different MATLAB tasks and compares the execution speed with the speed of several other computers. The six tasks are:

Test	Description	Performance Factors
LU	Perform LU of a full matrix	Floating-point, regular memory access
FFT	Perform FFT of a full vector	Floating-point, irregular memory access
ODE	Solve van der Pol equation with ODE45	Data structures and MATLAB function files
Sparse	Solve a symmetric sparse linear system	Mixed integer and floating-point
2-D	Plot Bernstein polynomial graph	2-D line drawing graphics
3-D	Display animated L-shape membrane logo	3-D animated OpenGL graphics

A final bar chart shows speed, which is inversely proportional to time. The longer bars represent faster machines, and the shorter bars represent the slower ones.

`bench(N)` runs each of the six tasks N times.

`bench(0)` just displays the results from other machines.

`t = bench(N)` returns an N-by-6 array with the execution times.

Tips

Note A benchmark is intended to compare performance of one particular version of MATLAB on different machines. It does not offer direct comparisons between different versions of MATLAB as tasks and problem sizes change from version to version.

The LU and FFT tasks involve large matrices and long vectors. The 2-D and 3-D tasks measure graphics performance, including software or hardware support for OpenGL. The command

```
opengl info
```

describes the OpenGL support available on a particular machine.

Fluctuations of five or ten percent in the measured times of repeated runs on a single machine are not uncommon. Your own mileage may vary.

See Also

`profile` | `profsave` | `mlint` | `mlintrpt` | `memory` | `pack` | `tic` | `cputime` | `rehash`

besselh

Purpose

Bessel function of third kind (Hankel function)

Note The behavior of `besselh` has changed in the following ways:

- The syntax `[H,ierr] = besselh(...)` has been removed.
- `besselh` no longer accepts `nu` and `Z` passed as a combination of row and column vectors.
- `besselh` no longer accepts `nu` and `Z` passed as a combination of non-scalar and empty inputs.

Syntax

```
H = besselh(nu,K,Z)
H = besselh(nu,Z)
H = besselh(nu,K,Z,1)
[H,ierr] = besselh(...)
```

Definitions

The differential equation

$$z^2 \frac{d^2 y}{dz^2} + z \frac{dy}{dz} + \left(z^2 - \nu^2\right) y = 0,$$

where ν is a nonnegative constant, is called *Bessel's equation*, and its solutions are known as *Bessel functions*. $J_\nu(z)$ and $J_{-\nu}(z)$ form a fundamental set of solutions of Bessel's equation for noninteger ν. $Y_\nu(z)$ is a second solution of Bessel's equation—linearly independent of $J_\nu(z)$—defined by

$$Y_\nu(z) = \frac{J_\nu(z)\cos(\nu\pi) - J_{-\nu}(z)}{\sin(\nu\pi)}.$$

The relationship between the Hankel and Bessel functions is

$$H_\nu^{(1)}(z) = J_\nu(z) + iY_\nu(z)$$
$$H_\nu^{(2)}(z) = J_\nu(z) - iY_\nu(z)$$

where $J_\nu(z)$ is `besselj`, and $Y_\nu(z)$ is `bessely`.

Description

`H = besselh(nu,K,Z)` computes the Hankel function $H_\nu^{(K)}(z)$ where `K` = 1 or 2, for each element of the complex array `Z`. If `nu` and `Z` are arrays of the same size, the result is also that size. If either input is a scalar, `besselh` expands it to the other input's size. If one input is a row vector and the other is a column vector, the result is a two-dimensional table of function values.

`H = besselh(nu,Z)` uses `K` = 1.

`H = besselh(nu,K,Z,1)` scales $H_\nu{}^{(K)}(z)$ by `exp(-i*Z)` if `K` = 1, and by `exp(+i*Z)` if `K` = 2.

`[H,ierr] = besselh(...)` also returns completion flags in an array the same size as `H`.

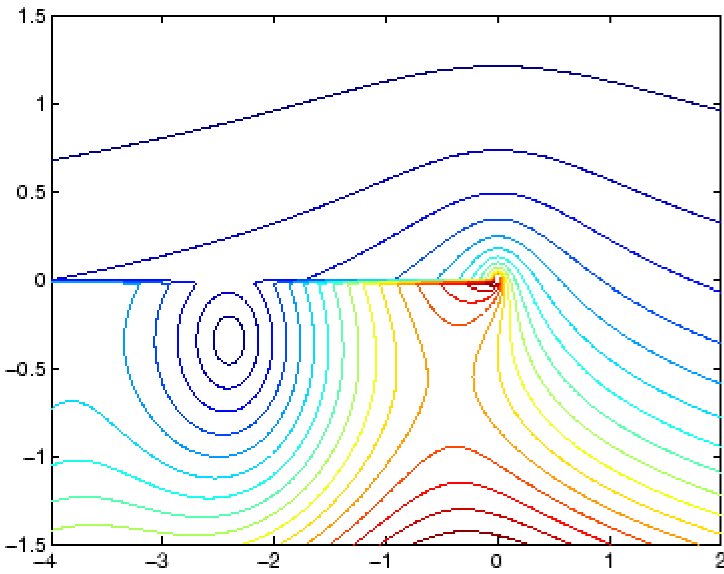
ierr	Description
0	`besselh` successfully computed the Hankel function for this element.
1	Illegal arguments.
2	Overflow. Returns `Inf`.
3	Some loss of accuracy in argument reduction.
4	Unacceptable loss of accuracy, `Z` or `nu` too large.
5	No convergence. Returns `NaN`.

Examples

This example generates the contour plots of the modulus and phase of the Hankel function $H_0{}^{(1)}(z)$ shown on page 359 of [1] Abramowitz and Stegun, *Handbook of Mathematical Functions*.

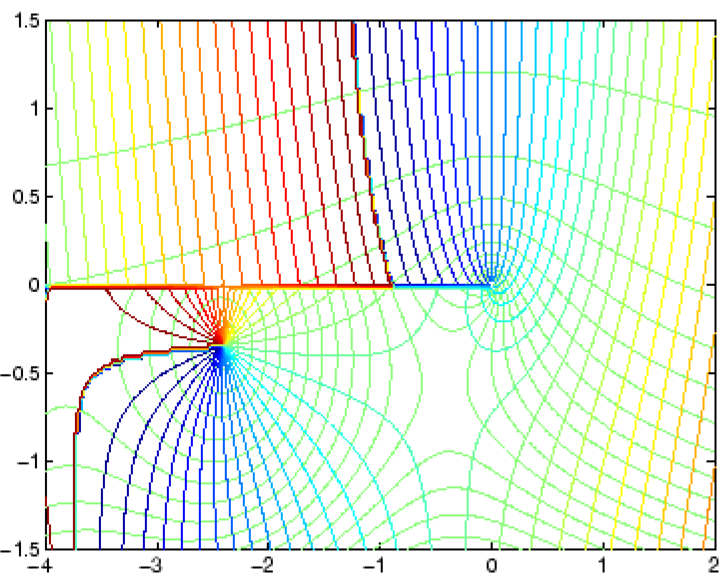
It first generates the modulus contour plot

```
[X,Y] = meshgrid(-4:0.025:2,-1.5:0.025:1.5);
H = besselh(0,1,X+i*Y);
contour(X,Y,abs(H),0:0.2:3.2), hold on
```



then adds the contour plot of the phase of the same function.

```
contour(X,Y,(180/pi)*angle(H),-180:10:180); hold off
```



References

[1] Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965.

See Also

`besselj` | `bessely` | `besseli` | `besselk`

besseli

Purpose

Modified Bessel function of first kind

Note The behavior of `besseli` has changed in the following ways:

- The syntax `[I,ierr] = besseli(...)` has been removed.
- `besseli` no longer accepts `nu` and `Z` passed as a combination of row and column vectors.
- `besseli` no longer accepts `nu` and `Z` passed as a combination of non-scalar and empty inputs.

Syntax

```
I = besseli(nu,Z)
I = besseli(nu,Z,1)
[I,ierr] = besseli(...)
```

Definitions

The differential equation

$$z^2 \frac{d^2y}{dz^2} + z\frac{dy}{dz} - \left(z^2 + \nu^2\right)y = 0,$$

where ν is a real constant, is called the *modified Bessel's equation*, and its solutions are known as *modified Bessel functions*.

$I_\nu(z)$ and $I_{-\nu}(z)$ form a fundamental set of solutions of the modified Bessel's equation for noninteger ν. $I_\nu(z)$ is defined by

$$I_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{(k=0)}^{\infty} \frac{\left(\frac{z^2}{4}\right)^k}{k!\Gamma(\nu + k + 1)}$$

where $\Gamma(a)$ is the gamma function.

$K_\nu(z)$ is a second solution, independent of $I_\nu(z)$. It can be computed using `besselk`.

Description

`I = besseli(nu,Z)` computes the modified Bessel function of the first kind, $I_\nu(z)$, for each element of the array `Z`. The order `nu` need not be an integer, but must be real. The argument `Z` can be complex. The result is real where `Z` is positive.

If `nu` and `Z` are arrays of the same size, the result is also that size. If either input is a scalar, it is expanded to the other input's size. If one input is a row vector and the other is a column vector, the result is a two-dimensional table of function values.

`I = besseli(nu,Z,1)` computes `besseli(nu,Z).*exp(-abs(real(Z)))`.

`[I,ierr] = besseli(...)` also returns completion flags in an array the same size as `I`.

ierr	Description
0	`besseli` successfully computed the modified Bessel function for this element.
1	Illegal arguments.
2	Overflow. Returns `Inf`.
3	Some loss of accuracy in argument reduction.
4	Unacceptable loss of accuracy, `Z` or `nu` too large.
5	No convergence. Returns `NaN`.

Examples

Example 1

```
format long
z = (0:0.2:1)';

besseli(1,z)

ans =
                   0
   0.10050083402813
```

```
0.20402675573357
0.31370402560492
0.43286480262064
0.56515910399249
```

Example 2

`besseli(3:9,(0:.2:10)',1)` generates the entire table on page 423 of [1] Abramowitz and Stegun, *Handbook of Mathematical Functions*

Algorithms

The `besseli` functions use a Fortran MEX-file to call a library developed by D.E. Amos [3] [4].

References

[1] Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sections 9.1.1, 9.1.89, and 9.12, formulas 9.1.10 and 9.2.5.

[2] Carrier, Krook, and Pearson, *Functions of a Complex Variable: Theory and Technique*, Hod Books, 1983, section 5.5.

[3] Amos, D.E., "A Subroutine Package for Bessel Functions of a Complex Argument and Nonnegative Order," *Sandia National Laboratory Report*, SAND85-1018, May, 1985.

[4] Amos, D.E., "A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order," *Trans. Math. Software*, 1986.

See Also

`airy` | `besselh` | `besselj` | `besselk` | `bessely`

Purpose

Bessel function of first kind

Note The behavior of `besselj` has changed in the following ways:

- The syntax `[J,ierr] = besselj(...)` has been removed.
- `besselj` no longer accepts `nu` and `Z` passed as a combination of row and column vectors.
- `besselj` no longer accepts `nu` and `Z` passed as a combination of non-scalar and empty inputs.

Syntax

```
J = besselj(nu,Z)
J = besselj(nu,Z,1)
[J,ierr] = besselj(nu,Z)
```

Definitions

The differential equation

$$z^2 \frac{d^2y}{dz^2} + z\frac{dy}{dz} + \left(z^2 - \nu^2\right)y = 0,$$

where ν is a real constant, is called *Bessel's equation*, and its solutions are known as *Bessel functions*.

$J_\nu(z)$ and $J_{-\nu}(z)$ form a fundamental set of solutions of Bessel's equation for noninteger ν. $J_\nu(z)$ is defined by

$$J_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{(k=0)}^{\infty} \frac{\left(\frac{-z^2}{4}\right)^k}{k!\,\Gamma(\nu + k + 1)}$$

where $\Gamma(a)$ is the gamma function.

$Y_\nu(z)$ is a second solution of Bessel's equation that is linearly independent of $J_\nu(z)$. It can be computed using `bessely`.

Description

`J = besselj(nu,Z)` computes the Bessel function of the first kind, $J_\nu(z)$, for each element of the array `Z`. The order `nu` need not be an integer, but must be real. The argument `Z` can be complex. The result is real where `Z` is positive.

If `nu` and `Z` are arrays of the same size, the result is also that size. If either input is a scalar, it is expanded to the other input's size. If one input is a row vector and the other is a column vector, the result is a two-dimensional table of function values.

`J = besselj(nu,Z,1)` computes `besselj(nu,Z).*exp(-abs(imag(Z)))`.

`[J,ierr] = besselj(nu,Z)` also returns completion flags in an array the same size as `J`.

ierr	Description
0	`besselj` successfully computed the Bessel function for this element.
1	Illegal arguments.
2	Overflow. Returns `Inf`.
3	Some loss of accuracy in argument reduction.
4	Unacceptable loss of accuracy, `Z` or `nu` too large.
5	No convergence. Returns `NaN`.

Tips

The Bessel functions are related to the Hankel functions, also called Bessel functions of the third kind,

$$H_\nu^{(1)}(z) = J_\nu(z) + iY_\nu(z)$$
$$H_\nu^{(2)}(z) = J_\nu(z) - iY_\nu(z)$$

where $H_\nu^{(K)}(z)$ is `besselh`, $J_\nu(z)$ is `besselj`, and $Y_\nu(z)$ is `bessely`. The Hankel functions also form a fundamental set of solutions to Bessel's equation (see `besselh`).

Examples

Example 1

```
format long
z = (0:0.2:1)';

besselj(1,z)

ans =
                    0
   0.09950083263924
   0.19602657795532
   0.28670098806392
   0.36884204609417
   0.44005058574493
```

Example 2

`besselj(3:9,(0:.2:10)')` generates the entire table on page 398 of [1] Abramowitz and Stegun, *Handbook of Mathematical Functions*.

Algorithms

The `besselj` function uses a Fortran MEX-file to call a library developed by D.E. Amos [3] [4].

References

[1] Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sections 9.1.1, 9.1.89, and 9.12, formulas 9.1.10 and 9.2.5.

[2] Carrier, Krook, and Pearson, *Functions of a Complex Variable: Theory and Technique*, Hod Books, 1983, section 5.5.

[3] Amos, D.E., “A Subroutine Package for Bessel Functions of a Complex Argument and Nonnegative Order,” *Sandia National Laboratory Report*, SAND85-1018, May, 1985.

[4] Amos, D.E., “A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order,” *Trans. Math. Software*, 1986.

See Also

`besselh` | `besseli` | `besselk` | `bessely`

Purpose

Modified Bessel function of second kind

Note The behavior of `besselk` has changed in the following ways:

- The syntax `[K,ierr] = besselk(...)` has been removed.
- `besselk` no longer accepts `nu` and `Z` passed as a combination of row and column vectors.
- `besselk` no longer accepts `nu` and `Z` passed as a combination of non-scalar and empty inputs.

Syntax

```
K = besselk(nu,Z)
K = besselk(nu,Z,1)
[K,ierr] = besselk(...)
```

Definitions

The differential equation

$$z^2 \frac{d^2 y}{dz^2} + z \frac{dy}{dz} - \left(z^2 + \nu^2\right) y = 0,$$

where ν is a real constant, is called the *modified Bessel's equation*, and its solutions are known as *modified Bessel functions*.

A solution $K_\nu(z)$ of the second kind can be expressed as:

$$K_\nu(z) = \left(\frac{\pi}{2}\right) \frac{I_{-\nu}(z) - I_\nu(z)}{\sin(\nu\pi)}$$

where $I_\nu(z)$ and $I_{-\nu}(z)$ form a fundamental set of solutions of the modified Bessel's equation for noninteger ν:

$$I_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} \frac{\left(\frac{z^2}{4}\right)^k}{k!\,\Gamma(\nu+k+1)}$$

and $\Gamma(a)$ is the gamma function. $K_\nu(z)$ is independent of $I_\nu(z)$.

$I_\nu(z)$ can be computed using `besseli`.

Description

`K = besselk(nu,Z)` computes the modified Bessel function of the second kind, $K_\nu(z)$, for each element of the array `Z`. The order `nu` need not be an integer, but must be real. The argument `Z` can be complex. The result is real where `Z` is positive.

If `nu` and `Z` are arrays of the same size, the result is also that size. If either input is a scalar, it is expanded to the other input's size. If one input is a row vector and the other is a column vector, the result is a two-dimensional table of function values.

`K = besselk(nu,Z,1)` computes `besselk(nu,Z).*exp(Z)`.

`[K,ierr] = besselk(...)` also returns completion flags in an array the same size as `K`.

ierr	Description
0	`besselk` successfully computed the modified Bessel function for this element.
1	Illegal arguments.
2	Overflow. Returns `Inf`.
3	Some loss of accuracy in argument reduction.
4	Unacceptable loss of accuracy, `Z` or `nu` too large.
5	No convergence. Returns `NaN`.

Examples

Example 1

```
format long
z = (0:0.2:1)';

besselk(1,z)

ans =
                Inf
   4.77597254322047
   2.18435442473269
   1.30283493976350
   0.86178163447218
   0.60190723019723
```

Example 2

`besselk(3:9,(0:.2:10)',1)` generates part of the table on page 424 of [1] Abramowitz and Stegun, *Handbook of Mathematical Functions*.

Algorithms

The `besselk` function uses a Fortran MEX-file to call a library developed by D.E. Amos [3], [4].

References

[1] Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sections 9.1.1, 9.1.89, and 9.12, formulas 9.1.10 and 9.2.5.

[2] Carrier, Krook, and Pearson, *Functions of a Complex Variable: Theory and Technique*, Hod Books, 1983, section 5.5.

[3] Amos, D.E., “A Subroutine Package for Bessel Functions of a Complex Argument and Nonnegative Order,” *Sandia National Laboratory Report*, SAND85-1018, May, 1985.

[4] Amos, D.E., “A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order,” *Trans. Math. Software*, 1986.

See Also

`airy` | `besselh` | `besseli` | `besselj` | `bessely`

Purpose

Bessel function of second kind

Note The behavior of `bessely` has changed in the following ways:

- The syntax `[Y,ierr] = bessely(...)` has been removed.
- `bessely` no longer accepts `nu` and `Z` passed as a combination of row and column vectors.
- `bessely` no longer accepts `nu` and `Z` passed as a combination of non-scalar and empty inputs.

Syntax

```
Y = bessely(nu,Z)
Y = bessely(nu,Z,1)
[Y,ierr] = bessely(nu,Z)
```

Definitions

The differential equation

$$z^2 \frac{d^2 y}{dz^2} + z \frac{dy}{dz} + \left(z^2 - \nu^2\right) y = 0,$$

where ν is a real constant, is called *Bessel's equation*, and its solutions are known as *Bessel functions*.

A solution $Y_\nu(z)$ of the second kind can be expressed as

$$Y_\nu(z) = \frac{J_\nu(z)\cos(\nu\pi) - J_{-\nu}(z)}{\sin(\nu\pi)}$$

where $J_\nu(z)$ and $J_{-\nu}(z)$ form a fundamental set of solutions of Bessel's equation for noninteger ν

$$J_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} \frac{\left(-\frac{z^2}{4}\right)^k}{k!\,\Gamma(\nu + k + 1)},$$

and $\Gamma(a)$ is the gamma function. $Y_\nu(z)$ is linearly independent of $J_\nu(z)$.

$J_\nu(z)$ can be computed using `besselj`.

Description

`Y = bessely(nu,Z)` computes Bessel functions of the second kind, $Y_\nu(z)$, for each element of the array `Z`. The order `nu` need not be an integer, but must be real. The argument `Z` can be complex. The result is real where `Z` is positive.

If `nu` and `Z` are arrays of the same size, the result is also that size. If either input is a scalar, it is expanded to the other input's size. If one input is a row vector and the other is a column vector, the result is a two-dimensional table of function values.

`Y = bessely(nu,Z,1)` computes `bessely(nu,Z).*exp(-abs(imag(Z)))`.

`[Y,ierr] = bessely(nu,Z)` also returns completion flags in an array the same size as `Y`.

ierr	Description
0	`bessely` successfully computed the Bessel function for this element.
1	Illegal arguments.
2	Overflow. Returns `Inf`.
3	Some loss of accuracy in argument reduction.
4	Unacceptable loss of accuracy, `Z` or `nu` too large.
5	No convergence. Returns `NaN`.

Tips

The Bessel functions are related to the Hankel functions, also called Bessel functions of the third kind,

$$H_\nu^{(1)}(z) = J_\nu(z) + i\,Y_\nu(z)$$
$$H_\nu^{(2)}(z) = J_\nu(z) - i\,Y_\nu(z),$$

where $H_{\nu}^{(K)}(z)$ is `besselh`, $J_{\nu}(z)$ is `besselj`, and $Y_{\nu}(z)$ is `bessely`. The Hankel functions also form a fundamental set of solutions to Bessel's equation (see `besselh`).

Examples

Example 1

```
format long
z = (0:0.2:1)';

bessely(1,z)

ans =
                  -Inf
  -3.32382498811185
  -1.78087204427005
  -1.26039134717739
  -0.97814417668336
  -0.78121282130029
```

Example 2

`bessely(3:9,(0:.2:10)')` generates the entire table on page 399 of [1] Abramowitz and Stegun, *Handbook of Mathematical Functions*.

Algorithms

The `bessely` function uses a Fortran MEX-file to call a library developed by D. E Amos [3] [4].

References

[1] Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sections 9.1.1, 9.1.89, and 9.12, formulas 9.1.10 and 9.2.5.

[2] Carrier, Krook, and Pearson, *Functions of a Complex Variable: Theory and Technique*, Hod Books, 1983, section 5.5.

[3] Amos, D.E., “A Subroutine Package for Bessel Functions of a Complex Argument and Nonnegative Order,” *Sandia National Laboratory Report*, SAND85-1018, May, 1985.

[4] Amos, D.E., “A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order,” *Trans. Math. Software*, 1986.

See Also

`besselh` | `besseli` | `besselj` | `besselk`

Purpose

Beta function

Syntax

```
B = beta(Z,W)
```

Definitions

The beta function is

$$B(z,w) = \int_0^1 t^{z-1}(1-t)^{w-1}\,dt = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)}$$

where $\Gamma(z)$ is the gamma function.

Description

`B = beta(Z,W)` computes the beta function for corresponding elements of arrays `Z` and `W`. The arrays must be real and nonnegative. They must be the same size, or either can be scalar.

Examples

In this example, which uses integer arguments,

```
beta(n,3)
   = (n-1)!*2!/(n+2)!
   = 2/(n*(n+1)*(n+2))
```

is the ratio of fairly small integers, and the rational format is able to recover the exact result.

```
format rat
beta((0:10)',3)

ans =

     1/0
     1/3
     1/12
     1/30
     1/60
     1/105
     1/168
     1/252
```

```
1/360
1/495
1/660
```

Algorithms

```
beta(z,w) = exp(gammaln(z)+gammaln(w)-gammaln(z+w))
```

See Also

`betainc` | `betaln` | `gammaln`

Purpose

Incomplete beta function

Syntax

```
I = betainc(X,Z,W)
I = betainc(X,Z,W,tail)
```

Definitions

The incomplete beta function is

$$I_x(z,w) = \frac{1}{B(z,w)} \int_0^x t^{z-1}(1-t)^{w-1} dt$$

where $B(z,w)$, the beta function, is defined as

$$B(z,w) = \int_0^1 t^{z-1}(1-t)^{w-1} dt = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)}$$

and $\Gamma(z)$ is the gamma function.

Description

`I = betainc(X,Z,W)` computes the incomplete beta function for corresponding elements of the arrays `X`, `Z`, and `W`. The elements of `X` must be in the closed interval [0,1]. The arrays `Z` and `W` must be nonnegative and real. All arrays must be the same size, or any of them can be scalar.

`I = betainc(X,Z,W,tail)` specifies the tail of the incomplete beta function. Choices are:

`'lower'` (the default)	Computes the integral from `0` to `x`
`'upper'`	Computes the integral from `x` to `1`

These functions are related as follows:

```
1-betainc(X,Z,W) = betainc(X,Z,W,'upper')
```

Note that especially when the upper tail value is close to `0`, it is more accurate to use the `'upper'` option than to subtract the `'lower'` value from `1`.

Examples

```
format long
betainc(.5,(0:10)',3)

ans =
   1.00000000000000
   0.87500000000000
   0.68750000000000
   0.50000000000000
   0.34375000000000
   0.22656250000000
   0.14453125000000
   0.08984375000000
   0.05468750000000
   0.03271484375000
   0.01928710937500
```

See Also

beta | betaln

betaincinv

Purpose

Beta inverse cumulative distribution function

Syntax

```
x = betaincinv(y,z,w)
x = betaincinv(y,z,w,tail)
```

Description

`x = betaincinv(y,z,w)` computes the inverse incomplete beta function for corresponding elements of `y`, `z`, and `w`, such that `y = betainc(x,z,w)`. The elements of `y` must be in the closed interval [0,1], and those of `z` and `w` must be nonnegative. `y`, `z`, and `w` must all be real and the same size (or any of them can be scalar).

`x = betaincinv(y,z,w,tail)` specifies the tail of the incomplete beta function. Choices are `lower` (the default) to use the integral from 0 to `x`, or `'upper'` to use the integral from `x` to 1. These two choices are related as follows: `betaincinv(y,z,w,'upper') = betaincinv(1-y,z,w,'lower')`. When `y` is close to 0, the `'upper'` option provides a way to compute `x` more accurately than by subtracting from `y` from 1.

Definitions

The incomplete beta function is defined as

$$I_x(z,w) = \frac{1}{\beta(z,w)} \int_0^x t^{(z-1)} (1-t)^{(w-1)} dt$$

`betaincinv` computes the inverse of the incomplete beta function with respect to the integration limit `x` using Newton's method.

See Also

`betainc` | `beta` | `betaln`

betaln

Purpose

Logarithm of beta function

Syntax

```
L = betaln(Z,W)
```

Description

`L = betaln(Z,W)` computes the natural logarithm of the `beta` function `log(beta(Z,W))`, for corresponding elements of arrays `Z` and `W`, without computing `beta(Z,W)`. Since the beta function can range over very large or very small values, its logarithm is sometimes more useful.

`Z` and `W` must be real and nonnegative. They must be the same size, or either can be scalar.

Examples

```
x = 510
betaln(x,x)

ans =
      -708.8616
```

`-708.8616` is slightly less than `log(realmin)`. Computing `beta(x,x)` directly would underflow (or be denormal).

Algorithms

```
betaln(z,w) = gammaln(z)+gammaln(w)-gammaln(z+w)
```

See Also

`beta` | `betainc` | `gammaln`

Purpose

Biconjugate gradients method

Syntax

```
x = bicg(A,b)
bicg(A,b,tol)
bicg(A,b,tol,maxit)
bicg(A,b,tol,maxit,M)
bicg(A,b,tol,maxit,M1,M2)
bicg(A,b,tol,maxit,M1,M2,x0)
[x,flag] = bicg(A,b,...)
[x,flag,relres] = bicg(A,b,...)
[x,flag,relres,iter] = bicg(A,b,...)
[x,flag,relres,iter,resvec] = bicg(A,b,...)
```

Description

`x = bicg(A,b)` attempts to solve the system of linear equations `A*x = b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle, `afun`, such that `afun(x,'notransp')` returns `A*x` and `afun(x,'transp')` returns `A'*x`.

"Parameterizing Functions" explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `bicg` converges, it displays a message to that effect. If `bicg` fails to converge after the maximum number of iterations or halts for any reason, it prints a warning message that includes the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`bicg(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `bicg` uses the default, `1e-6`.

`bicg(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `bicg` uses the default, `min(n,20)`.

`bicg(A,b,tol,maxit,M)` and `bicg(A,b,tol,maxit,M1,M2)` use the preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `bicg` applies

no preconditioner. M can be a function handle mfun, such that mfun(x,'notransp') returns M\x and mfun(x,'transp') returns M'\x.

bicg(A,b,tol,maxit,M1,M2,x0) specifies the initial guess. If x0 is [], then bicg uses the default, an all-zero vector.

[x,flag] = bicg(A,b,...) also returns a convergence flag.

Flag	Convergence
0	bicg converged to the desired tolerance tol within maxit iterations.
1	bicg iterated maxit times but did not converge.
2	Preconditioner M was ill-conditioned.
3	bicg stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during bicg became too small or too large to continue computing.

Whenever flag is not 0, the solution x returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the flag output is specified.

[x,flag,relres] = bicg(A,b,...) also returns the relative residual norm(b-A*x)/norm(b). If flag is 0, relres <= tol.

[x,flag,relres,iter] = bicg(A,b,...) also returns the iteration number at which x was computed, where 0 <= iter <= maxit.

[x,flag,relres,iter,resvec] = bicg(A,b,...) also returns a vector of the residual norms at each iteration including norm(b-A*x0).

Examples

Using bicg with a Matrix Input.

This example shows how to use bicg with a matrix input. bicg. The following code:

```
n = 100;
on = ones(n,1);
```

```
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);

x = bicg(A,b,tol,maxit,M1,M2);
```

displays this message:

```
bicg converged at iteration 9 to a solution with relative
residual 5.3e-009
```

Using bicg with a Function Handle

This example replaces the matrix `A` in the previous example with a handle to a matrix-vector product function `afun`. The example is contained in a file `run_bicg` that

- Calls `bicg` with the `@afun` function handle as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_bicg` are available to `afun`.

Place the following into a file called `run_bicg`:

```
function x1 = run_bicg
n = 100;
on = ones(n,1);
b = afun(on,'notransp');
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x1 = bicg(@afun,b,tol,maxit,M1,M2);

    function y = afun(x,transp_flag)
       if strcmp(transp_flag,'transp')      % y = A'*x
          y = 4 * x;
```

```
            y(1:n-1) = y(1:n-1) - 2 * x(2:n);
            y(2:n) = y(2:n) - x(1:n-1);
         elseif strcmp(transp_flag,'notransp') % y = A*x
            y = 4 * x;
            y(2:n) = y(2:n) - 2 * x(1:n-1);
            y(1:n-1) = y(1:n-1) - x(2:n);
         end
      end
   end
```

When you enter

```
x1 = run_bicg;
```

MATLAB software displays the message

```
bicg converged at iteration 9 to a solution with ...
relative residual
5.3e-009
```

Using a Preconditioner

This example demonstrates the use of a preconditioner.

1

Load `A = west0479`, a real 479-by-479 nonsymmetric sparse matrix:

```
load west0479;
A = west0479;
```

2

Define `b` so that the true solution is a vector of all ones:

```
b = full(sum(A,2));
```

3

Set the tolerance and maximum number of iterations:

```
tol = 1e-12; maxit = 20;
```

4

Use `bicg` to find a solution at the requested tolerance and number of iterations:

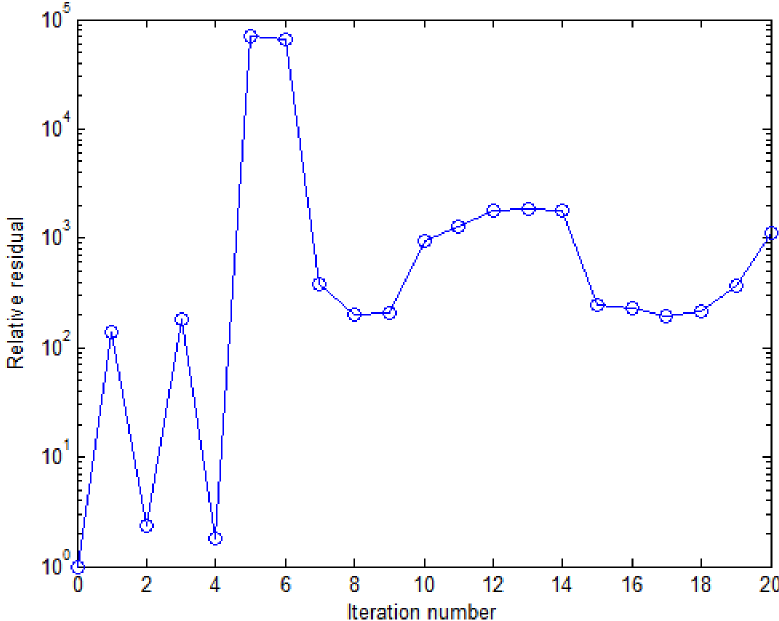
```
[x0,fl0,rr0,it0,rv0] = bicg(A,b,tol,maxit);
```

`fl0` is 1 because `bicg` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. In fact, the behavior of `bicg` is so poor that the initial guess (`x0 = zeros(size(A,2),1)`) is the best solution and is returned as indicated by `it0 = 0`. MATLAB stores the residual history in `rv0`.

5

Plot the behavior of `bicg`:

```
semilogy(0:maxit,rv0/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



The plot shows that the solution does not converge. You can use a preconditioner to improve the outcome.

6

Create the preconditioner with `ilu`, since the matrix `A` is nonsymmetric:

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));
Error using ilu
There is a pivot equal to zero. Consider decreasing
the drop tolerance or consider using the 'udiag' option.
```

MATLAB cannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

7

You can try again with a reduced drop tolerance, as indicated by the error message:

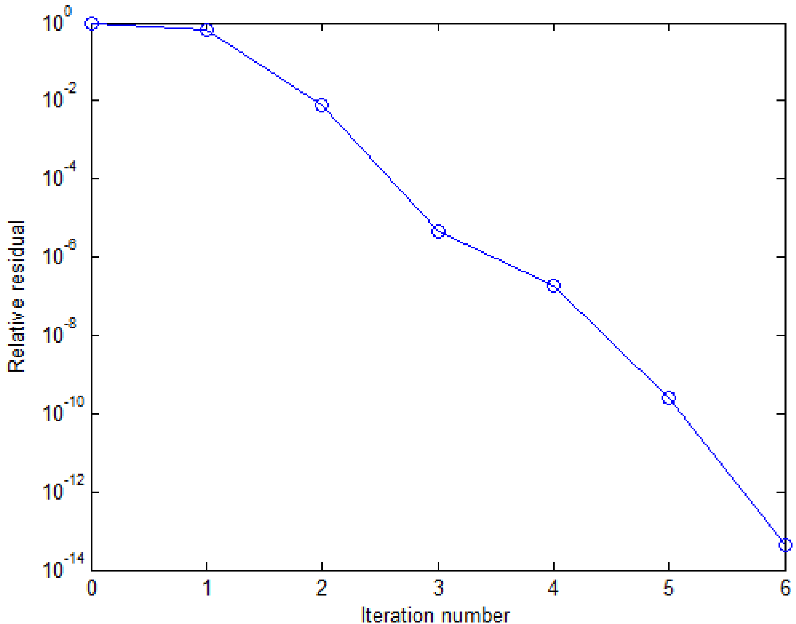
```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = bicg(A,b,tol,maxit,L,U);
```

`fl1` is 0 because `bicg` drives the relative residual to `4.1410e-014` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the sixth iteration (the value of `it1`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output `rv1(1)` is `norm(b)`, and the output `rv1(7)` is `norm(b-A*x2)`.

8

You can follow the progress of `bicg` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0):

```
semilogy(0:it1,rv1/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



References

[1] Barrett, R., M. Berry, T.F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

See Also

`bicgstab` | `cgs` | `gmres` | `ilu` | `lsqr` | `luinc` | `minres` | `pcg` | `qmr` | `symmlq` | `function_handle` | `mldivide`

Purpose

Biconjugate gradients stabilized method

Syntax

```
x = bicgstab(A,b)
bicgstab(A,b,tol)
bicgstab(A,b,tol,maxit)
bicgstab(A,b,tol,maxit,M)
bicgstab(A,b,tol,maxit,M1,M2)
bicgstab(A,b,tol,maxit,M1,M2,x0)
[x,flag] = bicgstab(A,b,...)
[x,flag,relres] = bicgstab(A,b,...)
[x,flag,relres,iter] = bicgstab(A,b,...)
[x,flag,relres,iter,resvec] = bicgstab(A,b,...)
```

Description

`x = bicgstab(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle, `afun`, such that `afun(x)` returns `A*x`.

"Parameterizing Functions" explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `bicgstab` converges, a message to that effect is displayed. If `bicgstab` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`bicgstab(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `bicgstab` uses the default, `1e-6`.

`bicgstab(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `bicgstab` uses the default, `min(n,20)`.

`bicgstab(A,b,tol,maxit,M)` and `bicgstab(A,b,tol,maxit,M1,M2)` use preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `bicgstab` applies no preconditioner. `M` can be a function handle `mfun`, such that `mfun(x)` returns `M\x`.

`bicgstab(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `bicgstab` uses the default, an all zero vector.

`[x,flag] = bicgstab(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	`bicgstab` converged to the desired tolerance `tol` within `maxit` iterations.
1	`bicgstab` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`bicgstab` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `bicgstab` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = bicgstab(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = bicgstab(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`. `iter` can be an integer + 0.5, indicating convergence halfway through an iteration.

`[x,flag,relres,iter,resvec] = bicgstab(A,b,...)` also returns a vector of the residual norms at each half iteration, including `norm(b-A*x0)`.

Examples

Using bicgstab with a Matrix Input

This example first solves `Ax = b` by providing `A` and the preconditioner `M1` directly as arguments.

The code:

```
A = gallery('wilk',21);
b = sum(A,2);
tol = 1e-12;
maxit = 15;
M1 = diag([10:-1:1 1 1:10]);

x = bicgstab(A,b,tol,maxit,M1);
```

displays the message:

```
bicgstab converged at iteration 12.5 to a solution with relative
residual 2e-014.
```

Using bicgstab with a Function Handle

This example replaces the matrix `A` in the previous example with a handle to a matrix-vector product function `afun`, and the preconditioner `M1` with a handle to a backsolve function `mfun`. The example is contained in a file `run_bicgstab` that

- Calls `bicgstab` with the function handle `@afun` as its first argument.
- Contains `afun` and `mfun` as nested functions, so that all variables in `run_bicgstab` are available to `afun` and `mfun`.

The following shows the code for `run_bicgstab`:

```
function x1 = run_bicgstab
n = 21;
b = afun(ones(n,1));
tol = 1e-12;
maxit = 15;
x1 = bicgstab(@afun,b,tol,maxit,@mfun);

    function y = afun(x)
       y = [0; x(1:n-1)] + ...
           [((n-1)/2:-1:0)'; (1:(n-1)/2)'].*x + ...
           [x(2:n); 0];
    end
```

```
    function y = mfun(r)
        y = r ./ [((n-1)/2:-1:1)'; 1; (1:(n-1)/2)'];
    end
end
```

When you enter

```
x1 = run_bicgstab;
```

MATLAB software displays the message

```
bicgstab converged at iteration 12.5 to a solution with relative
residual 2e-014.
```

Using a Preconditioner

This example demonstrates the use of a preconditioner.

1

Load `west0479`, a real 479-by-479 nonsymmetric sparse matrix:

```
load west0479;
A = west0479;
```

2

Define `b` so that the true solution is a vector of all ones:

```
b = full(sum(A,2));
```

3

Set the tolerance and maximum number of iterations:

```
tol = 1e-12; maxit = 20;
```

4

Use `bicgstab` to find a solution at the requested tolerance and number of iterations:

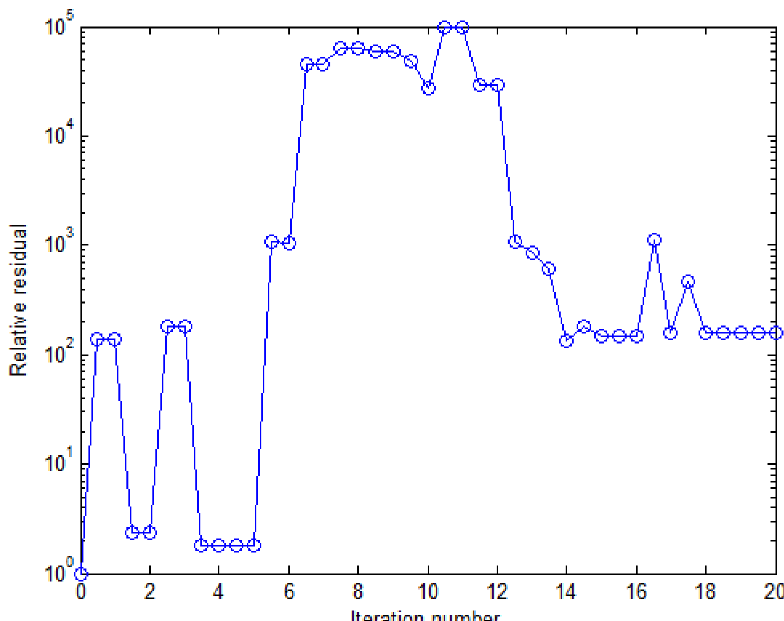
```
[x0,fl0,rr0,it0,rv0] = bicgstab(A,b,tol,maxit);
```

`fl0` is 1 because `bicgstab` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. In fact, the behavior of `bicgstab` is so bad that the initial guess (`x0 = zeros(size(A,2),1)`) is the best solution and is returned as indicated by `it0 = 0`. MATLAB stores the residual history in `rv0`.

5

Plot the behavior of `bicgstab`:

```
semilogy(0:0.5:maxit,rv0/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



The plot shows that the solution does not converge. You can use a preconditioner to improve the outcome.

6

Create a preconditioner with `ilu`, since `A` is nonsymmetric:

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));
Error using ilu
There is a pivot equal to zero. Consider decreasing
the drop tolerance or consider using the 'udiag' option.
```

MATLAB cannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

7

You can try again with a reduced drop tolerance, as indicated by the error message:

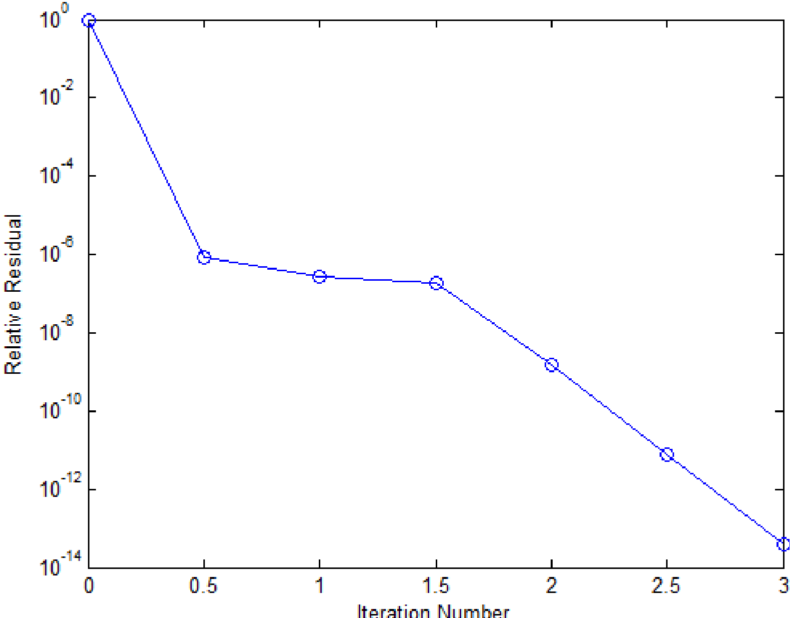
```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = bicgstab(A,b,tol,maxit,L,U);
```

`fl1` is 0 because `bicgstab` drives the relative residual to `5.9829e-014` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the third iteration (the value of `it1`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output `rv1(1)` is `norm(b)`and the output `rv1(7)` is `norm(b-A*x2)` since `bicgstab` uses half iterations.

8

You can follow the progress of `bicgstab` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0):

```
semilogy(0:0.5:it1,rv1/norm(b),'-o');
xlabel('Iteration Number');
ylabel('Relative Residual');
```



References

[1] Barrett, R., M. Berry, T.F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] van der Vorst, H.A., "BI-CGSTAB: A fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, March 1992, Vol. 13, No. 2, pp. 631–644.

See Also

`bicg` | `cgs` | `gmres` | `lsqr` | `luinc` | `minres` | `pcg` | `qmr` | `symmlq` | `function_handle` | `mldivide`

bicgstabl

Purpose

Biconjugate gradients stabilized (l) method

Syntax

```
x = bicgstabl(A,b)
x = bicgstabl(afun,b)
x = bicgstabl(A,b,tol)
x = bicgstabl(A,b,tol,maxit)
x = bicgstabl(A,b,tol,maxit,M)
x = bicgstabl(A,b,tol,maxit,M1,M2)
x = bicgstabl(A,b,tol,maxit,M1,M2,x0)
[x,flag] = bicgstabl(A,b,...)
[x,flag,relres] = bicgstabl(A,b,...)
[x,flag,relres,iter] = bicgstabl(A,b,...)
[x,flag,relres,iter,resvec] = bicgstabl(A,b,...)
```

Description

`x = bicgstabl(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and the right-hand side column vector `b` must have length `n`.

`x = bicgstabl(afun,b)` accepts a function handle `afun` instead of the matrix `A`. `afun(x)` accepts a vector input `x` and returns the matrix-vector product `A*x`. In all of the following syntaxes, you can replace `A` by `afun`.

`x = bicgstabl(A,b,tol)` specifies the tolerance of the method. If `tol` is [] then `bicgstabl` uses the default, 1e-6.

`x = bicgstabl(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is [] then `bicgstabl` uses the default, `min(N,20)`.

`x = bicgstabl(A,b,tol,maxit,M)` and `x = bicgstabl(A,b,tol,maxit,M1,M2)` use preconditioner `M` or `M=M1*M2` and effectively solve the system `A*inv(M)*x = b` for x. If `M` is [] then a preconditioner is not applied. `M` may be a function handle returning `M\x`.

`x = bicgstabl(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is [] then `bicgstabl` uses the default, an all zero vector.

`[x,flag] = bicgstabl(A,b,...)` also returns a convergence `flag`:

Flag	Convergence
0	`bicgstabl` converged to the desired tolerance `tol` within `maxit` iterations.
1	`bicgstabl` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`bicgstabl` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `bicgstabl` became too small or too large to continue computing.

`[x,flag,relres] = bicgstabl(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = bicgstabl(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`. `iter` can be `k/4` where `k` is some integer, indicating convergence at a given quarter iteration.

`[x,flag,relres,iter,resvec] = bicgstabl(A,b,...)` also returns a vector of the residual norms at each quarter iteration, including `norm(b-A*x0)`.

Examples

Using bicgstabl with Inputs or with a Function

You can pass inputs directly to `bicgstabl`:

```
n = 21;
A = gallery('wilk',n);
b = sum(A,2);
tol = 1e-12;
maxit = 15;
M = diag([10:-1:1 1 1:10]);
x = bicgstabl(A,b,tol,maxit,M);
```

You can also use a matrix-vector product function:

```
function y = afun(x,n)
y = [0; x(1:n-1)] + [((n-1)/2:-1:0)';
(1:(n-1)/2)'].*x+[x(2:n); 0];
```

and a preconditioner backsolve function:

```
function y = mfun(r,n)
y = r ./ [((n-1)/2:-1:1)';
1;
(1:(n-1)/2)'];
```

as inputs to `bicgstabl`:

```
x1 = bicgstabl(@(x)afun(x,n),b,tol,maxit,@(x)mfun(x,n));
```

Using a Preconditioner

This example demonstrates the use of a preconditioner.

1

Load `west0479`, a real 479-by-479 nonsymmetric sparse matrix:

```
load west0479;
A = west0479;
```

2

Define `b` so that the true solution is a vector of all ones.

```
b = full(sum(A,2));
```

3

Set the tolerance and maximum number of iterations:

```
tol = 1e-12; maxit = 20;
```

4

Use `bicgstabl` to find a solution at the requested tolerance and number of iterations:

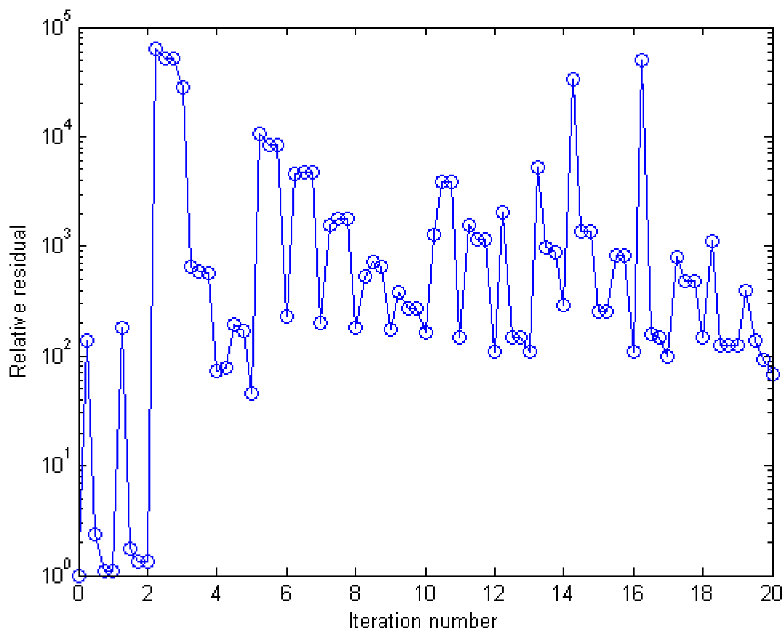
```
[x0,fl0,rr0,it0,rv0] = bicgstabl(A,b,tol,maxit);
```

`fl0` is 1 because `bicgstabl` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. In fact, the behavior of `bicgstabl` is so poor that the initial guess (`x0 = zeros(size(A,2),1)`) is the best solution and is returned as indicated by `it0 = 0`. MATLAB stores the residual history in `rv0`.

5

Plot the behavior of `bicgstabl`:

```
semilogy(0:0.25:maxit,rv0/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



The plot shows that the solution does not converge. You can use a preconditioner to improve the outcome.

6

Create a preconditioner with `ilu`, since `A` is nonsymmetric:

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));
Error using ilu
There is a pivot equal to zero.  Consider decreasing the
drop tolerance or consider using the 'udiag' option.
```

MATLAB cannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

7

You can try again with a reduced drop tolerance, as indicated by the error message:

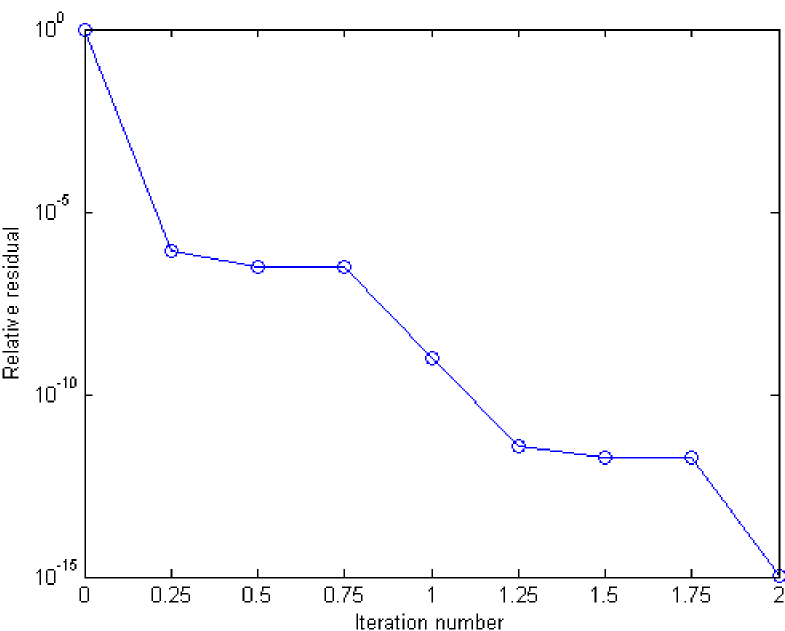
```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = bicgstabl(A,b,tol,maxit,L,U);
```

`fl1` is 0 because `bicgstabl` drives the relative residual to `1.0257e-015` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the sixth iteration (the value of `it1`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output `rv1(1)` is `norm(b)`, and the output `rv1(9)` is `norm(b-A*x2)` since `bicgstabl` uses quarter iterations.

8

You can follow the progress of `bicgstabl` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0):

```
semilogy(0:0.25:it1,rv1/norm(b),'-o');
set(gca,'XTick',0:0.25:it1);
xlabel('Iteration number');
ylabel('Relative residual');
```



See Also

`bicgstab` | `bicg` | `cgs` | `gmres` | `lsqr` | `luinc` | `minres` | `pcg` | `qmr` | `symmlq` | `function_handle` | `mldivide`

bin2dec

Purpose

Convert binary number string to decimal number

Syntax

```
bin2dec(binarystr)
```

Description

`bin2dec(binarystr)` interprets the binary string *binarystr* and returns the equivalent decimal number.

`bin2dec` ignores any space (`' '`) characters in the input string.

Examples

Binary `010111` converts to decimal `23`:

```
bin2dec('010111')
ans =
    23
```

Because space characters are ignored, this string yields the same result:

```
bin2dec(' 010   111 ')
ans =
    23
```

See Also

`dec2bin`

Purpose

Set FTP transfer type to binary

Syntax

```
binary(ftpobj)
```

Description

`binary(ftpobj)` sets the FTP download and upload mode to binary, which does not convert new line characters. Binary mode is the default for FTP objects. If you previously called the `ascii` method, use this method before transferring a nontext file, such as an executable or ZIP archive.

Input Arguments

ftpobj

FTP object created by `ftp`.

Examples

Connect to the MathWorks FTP server, and set the transfer mode to binary:

```
mw=ftp('ftp.mathworks.com');
binary(mw)
```

See Also

`ascii` | `ftp`

bitand

Purpose

Bit-wise AND

Syntax

```
intout = bitand(integ1,integ2)
intout = bitand(integ1,integ2,assumedtype)

objout = bitand(netobj1,netobj2)
```

Description

`intout = bitand(integ1,integ2)` returns the bit-wise AND of values `integ1` and `integ2`.

`intout = bitand(integ1,integ2,assumedtype)` assumes that `integ1` and `integ2` are of `assumedtype`.

`objout = bitand(netobj1,netobj2)` returns the bit-wise AND of the .NET enumeration objects `netobj1` and `netobj2`.

Input Arguments

integ1, integ2 - Input values

signed integer arrays | unsigned integer arrays | double arrays

Input values, specified as signed integer arrays, unsigned integer arrays, or double arrays. `integ1` and `integ2` must be the same data type, or one must be a scalar double value.

- If `integ1` and `integ2` are double arrays, and `assumedtype` is not specified, then MATLAB treats `integ1` and `integ2` as unsigned 64-bit integers.
- If `assumedtype` is specified, then all elements in `integ1` and `integ2` must have integer values within the range of `assumedtype`.

Data Types

`double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

assumedtype - Assumed data type of `integ1` and `integ2`

`'uint64' | 'uint32' | 'uint16' | 'uint8' | 'int64' | 'int32' | 'int16' | 'int8'`

Assumed data type of `integ1` and `integ2`, specified as `'uint64'`, `'uint32'`, `'uint16'`, `'uint8'`, `'int64'`, `'int32'`, `'int16'`, or `'int8'`.

- If `integ1` and `integ2` are double arrays, then `assumedtype` can specify any valid integer type, but defaults to `'uint64'`.
- If `integ1` and `integ2` are integer type arrays, then `assumedtype` must specify that same integer type.

Data Types
`char`

netobj1, netobj2 - Input values

.NET enumeration objects

Input values, specified as .NET enumeration objects. You must be running a version of Windows to use .NET enumeration objects as input arguments.

Output Arguments

intout - Bit-wise AND result

signed integer array | unsigned integer array | double array

Bit-wise AND result, returned as a signed integer array, unsigned integer array, or double array. `intout` is the same data type and size as `integ1` and `integ2`.

- If either `integ1` or `integ2` is a scalar double, and the other is a non-double integer type, `intout` is the non-double integer type.

objout - Bit-wise AND result

.NET enumeration object

Bit-wise AND result, returned as a .NET enumeration objects.

Examples

Truth Table

Create a truth table for the logical AND operation.

```
A = uint8([0 1; 0 1]);
B = uint8([0 0; 1 1]);
TTable = bitand(A, B)
```

```
TTable =

     0     0
     0     1
```

`bitand` returns 1 only if both bit-wise inputs are 1.

Negative Values

Explore how `bitand` handles negative values

MATLAB encodes signed integers using two's complement. Thus, the bit-wise AND of 5 (11111010) and 6 (00000110) is 2 (00000010).

```
C = -5;
D = 6;
bitand(C,D,'int8')
```

```
ans =

      2
```

See Also

bitcmp | bitget | bitor | bitshift | bitset | bitxor | intmax

Related Examples

- "Creating .NET Enumeration Bit Flags"

Purpose

Bit-wise complement

Compatibility

`bitcmp(A,N)` will not accept N in a future release. Use `bitcmp(A)` or `bitcmp(A,assumedtype)` instead.

Syntax

```
cmp = bitcmp(A)
cmp = bitcmp(A,assumedtype)

cmp = bitcmp(A,N)
```

Description

`cmp = bitcmp(A)` returns the bit-wise complement of A.

`cmp = bitcmp(A,assumedtype)` assumes that A is of `assumedtype`.

`cmp = bitcmp(A,N)` returns an N-bit complement of A. Elements of A cannot exceed 2^N-1.

Input Arguments

A - Input value

signed integer array | unsigned integer array | double array

Input value, specified as a signed integer array, unsigned integer array, or double array.

- If A is a double array, and `assumedtype` is not specified, then MATLAB treats A as an unsigned 64-bit integer.
- If `assumedtype` is specified, then all elements in A must have integer values within the range of `assumedtype`.

Data Types

`double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

assumedtype - Assumed data type of A

`'uint64' | 'uint32' | 'uint16' | 'uint8' | 'int64' | 'int32' | 'int16' | 'int8'`

Assumed data type of `A`, specified as `'uint64'`, `'uint32'`, `'uint16'`, `'uint8'`, `'int64'`, `'int32'`, `'int16'`, or `'int8'`.

- If `A` is a double array, then `assumedtype` can specify any valid integer type, but defaults to `'uint64'`.
- If `A` is an integer type array, then `assumedtype` must specify that same integer type.

Data Types
`char`

N - number of returned bits

integer

Number of returned bits, specified as an integer. N cannot exceed the number of bits in the integer type of `A`.

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

Output Arguments

cmp - Bit-wise complement

signed integer array | unsigned integer array | double array

Bit-wise complement, returned as a signed integer array, unsigned integer array, or double array. `cmp` is the same size and type as `A`.

Examples

Complement of a Negative Integer

```
A = int8(-11);
cmp = bitcmp(A)
```

```
cmp =

   10
```

You can see the complement operation when the numbers are shown in binary.

```
original = bitget(A,8:-1:1)
complement = bitget(bitcmp(A),8:-1:1)

original =

     1     1     1     1     0     1     0     1


complement =

     0     0     0     0     1     0     1     0
```

Complement of Unsigned Integers

```
cmp = bitcmp(64,'uint8')
maxint = intmax('uint8') - 64

cmp =

   191


maxint =

  191
```

The complement of an unsigned integer is equal to itself subtracted from the maximum integer of its data type.

See Also

bitand | bitget | bitor | bitshift | bitset | bitxor | intmax

bitget

Purpose

Get bit at specified position

Syntax

```
b = bitget(A,bit)
b = bitget(A,bit,assumedtype)
```

Description

`b = bitget(A,bit)` returns the bit value at position `bit` in integer array `A`.

`b = bitget(A,bit,assumedtype)` assumes that `A` is of `assumedtype`.

Input Arguments

A - Input value

signed integer array | unsigned integer array | double array

Input value, specified as a signed integer array, unsigned integer array, or double array.

- If `A` is a double array, and `assumedtype` is not specified, then MATLAB treats `A` as an unsigned 64-bit integer.
- If `assumedtype` is specified, then all elements in `A` must have integer values within the range of `assumedtype`.

Data Types

`double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

bit - Bit position

integer | integer array

Bit position, specified as an integer or integer array. `bit` must be between 1 (the least-significant bit) and the number of bits in the integer class of `A`.

Data Types

`double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

assumedtype - Assumed data type of A

'uint64' | 'uint32' | 'uint16' | 'uint8' | 'int64' | 'int32' | 'int16' | 'int8'

Assumed data type of A, specified as 'uint64', 'uint32', 'uint16', 'uint8', 'int64', 'int32', 'int16', or 'int8'.

- If A is a double array, then assumedtype can specify any valid integer type, but defaults to 'uint64'.
- If A is an integer type array, then assumedtype must specify that same integer type.

Data Types
char

Output Arguments

b - Bit value at bit

0 | 1

Bit value at bit, returned as 0 or 1. b is the same data type as A.

Examples

Maximum Integer

Find the difference in the binary representation between the maximum integer of signed and unsigned integers.

```
a1 = intmax('int8');
a2 = intmax('uint8');
b1 = bitget(a1,8:-1:1)
b2 = bitget(a2,8:-1:1)
```

```
b1 =

     0     1     1     1     1     1     1     1


b2 =

     1     1     1     1     1     1     1     1
```

The signed integers require a bit to accommodate negative integers.

Negative Numbers Using Two's Complement

Find the 8-bit representation of a negative number.

```
A = -29;
b = bitget(A,8:-1:1,'int8')

b =

     1     1     1     0     0     0     1     1
```

See Also

bitand | bitcmp | bitor | bitshift | bitset | bitxor | intmax

Purpose

Maximum double-precision floating-point integer

Compatibility

`bitmax` will be removed in a future release. Use `flintmax` instead.

Syntax

```
bitmax
```

Description

`bitmax` returns the maximum unsigned double-precision floating-point integer for your computer. It is the value when all bits are set, namely the value $2^{53} - 1$.

Note Instead of integer-valued double-precision variables, use unsigned integers for bit manipulations and replace `bitmax` with `intmax`.

Examples

Display in different formats the largest floating point integer and the largest 32 bit unsigned integer:

```
format long e
bitmax
ans =
   9.007199254740991e+015

intmax('uint32')
ans =
   4294967295

format hex
bitmax
ans =
   433fffffffffffff

intmax('uint32')
ans =
   ffffffff
```

In the second bitmax statement, the last 13 hex digits of `bitmax` are `f`, corresponding to 52 1's (all 1's) in the mantissa of the binary representation. The first 3 hex digits correspond to the sign bit 0 and the 11 bit biased exponent `10000110011` in binary (`1075` in decimal), and the actual exponent is (`1075-1023`) = `52`. Thus the binary value of `bitmax` is `1.111...111 x 2^52` with 52 trailing 1's, or `2^53-1`.

See Also

`bitand` | `bitcmp` | `bitget` | `bitor` | `bitset` | `bitshift` | `bitxor`

Purpose

Bit-wise NOT

Syntax

```
C = bitnot(A)
```

Description

`C = bitnot(A)` reverses all bits.

Input Arguments

A

.NET enumeration object.

Output Arguments

C

Result of operation, same type as input.

See Also

`bitand` | `bitor` | `bitxor`

bitor

Purpose

Bit-wise OR

Syntax

```
intout = bitor(integ1,integ2)
intout = bitor(integ1,integ2,assumedtype)

objout = bitor(netobj1,netobj2)
```

Description

`intout = bitor(integ1,integ2)` returns the bit-wise OR of `integ1` and `integ2`.

`intout = bitor(integ1,integ2,assumedtype)` assumes that `integ1` and `integ2` are of `assumedtype`.

`objout = bitor(netobj1,netobj2)` returns the bit-wise OR of the .NET enumeration objects `netobj1` and `netobj2`.

Input Arguments

integ1, integ2 - Input values

signed integer arrays | unsigned integer arrays | double arrays

Input values, specified as signed integer arrays, unsigned integer arrays, or double arrays. `integ1` and `integ2` must be the same data type, or one must be a scalar double value.

- If `integ1` and `integ2` are double arrays, and `assumedtype` is not specified, then MATLAB treats `integ1` and `integ2` as unsigned 64-bit integers.
- If `assumedtype` is specified, then all elements in `integ1` and `integ2` must have integer values within the range of `assumedtype`.

Data Types

`double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

assumedtype - Assumed data type of `integ1` and `integ2`

`'uint64' | 'uint32' | 'uint16' | 'uint8' | 'int64' | 'int32' | 'int16' | 'int8'`

Assumed data type of `integ1` and `integ2`, specified as `'uint64'`, `'uint32'`, `'uint16'`, `'uint8'`, `'int64'`, `'int32'`, `'int16'`, or `'int8'`.

- If `integ1` and `integ2` are double arrays, then `assumedtype` can specify any valid integer type, but defaults to `'uint64'`.
- If `integ1` and `integ2` are integer type arrays, then `assumedtype` must specify that same integer type.

Data Types
`char`

netobj1, netobj2 - Input values

.NET enumeration objects

Input values, specified as .NET enumeration objects. You must be running a version of Windows to use .NET enumeration objects as input arguments.

Output Arguments

intout - Bit-wise OR result

signed integer array | unsigned integer array | double array

Bit-wise OR result, returned as a signed integer array, unsigned integer array, or double array. `intout` is the same data type and size as `integ1` and `integ2`.

- If either `integ1` or `integ2` is a scalar double, and the other is a non-double integer type, `intout` is the non-double integer type.

objout - Bit-wise OR result

.NET enumeration object

Bit-wise OR result, returned as a .NET enumeration objects.

Examples

Truth Table

Create a truth table for the logical OR operation.

```
A = uint8([0 1; 0 1]);
B = uint8([0 0; 1 1]);
TTable = bitor(A, B)
```

```
TTable =

     0     1
     1     1
```

`bitor` returns 1 if either bit-wise input is 1.

Negative Values

Explore how `bitor` handles negative values.

MATLAB encodes negative integers using two's complement. Thus, the bit-wise OR of -5 (11111010) and 6 (00000110) is -1 (11111110).

```
C = -5;
D = 6;
bitor(C,D,'int8')

ans =

    -1
```

See Also

`bitand` | `bitcmp` | `bitget` | `bitshift` | `bitset` | `bitxor` | `intmax`

Related Examples

- "Creating .NET Enumeration Bit Flags"

Purpose

Set bit at specific location

Syntax

```
intout = bitset(A,bit)
intout = bitset(A,bit,assumedtype)

intout = bitset(A,bit,V)
intout = bitset(A,bit,V,assumedtype)
```

Description

`intout = bitset(A,bit)` returns the value of `A` with position `bit` set to 1 (on).

`intout = bitset(A,bit,assumedtype)` assumes `A` is of type `assumedtype`.

`intout = bitset(A,bit,V)` returns `A` with position `bit` set to the value of `V`.

- If `V` is zero, then the bit position `bit` is set to 0 (off).
- If `V` is nonzero, then the bit position `bit` is set to 1 (on).

`intout = bitset(A,bit,V,assumedtype)` assumes `A` is of type `assumedtype`.

Input Arguments

A - Input value

signed integer array | unsigned integer array | double array

Input value, specified as a signed integer array, unsigned integer array, or double array.

- If `A` is a double array, and `assumedtype` is not specified, then MATLAB treats `A` as an unsigned 64-bit integer.
- If `assumedtype` is specified, then all elements in `A` must have integer values within the range of `assumedtype`.

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

bit - Bit position

integer | integer array

Bit position, specified as an integer or integer array. `bit` must be between 1 (the least significant bit) and the number of bits in the integer class of `A`.

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

assumedtype - Assumed data type of A

`'uint64'` | `'uint32'` | `'uint16'` | `'uint8'` | `'int64'` | `'int32'` | `'int16'` | `'int8'`

Assumed data type of `A`, specified as `'uint64'`, `'uint32'`, `'uint16'`, `'uint8'`, `'int64'`, `'int32'`, `'int16'`, or `'int8'`.

- If `A` is a double array, then `assumedtype` can specify any valid integer type, but defaults to `'uint64'`.
- If `A` is an integer type array, then `assumedtype` must specify that same integer type.

Data Types
`char`

V - bit value

scalar | numeric array

Bit value, specified as a scalar or a numeric array. If `V` and `bit` are arrays, they must be the same size.

Data Types
`double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Output Arguments

intout - Adjusted integer

signed integer array | unsigned integer array | double array

Adjusted integer, returned as a signed integer array, unsigned integer array, or double array. `intout` is the same size and type as `A`.

Examples

Set Bits to On

Add powers of 2 onto a number.

```
A = 4;
intout = bitset(A,4:6)

intout =

    12    20    36
```

You can see that `bitset` sequentially turns on bits 4 through 6.

```
c = dec2bin(intout)

c =

001100
010100
100100
```

Out of Range of Integer Type

MATLAB throws an error if you specify an integer outside the range of `assumedtype`.

```
intout = bitset(300,5,'int8')

Error using bitset
Double inputs must have integer values in the range of ASSUMEDTYPE.
```

You can avoid this error by limiting your input to the range of the specified datatype.

Set Bits to Off

Repeatedly subtract powers of 2 from a number.

```
a = intmax('uint8')
for k = 0:7
   a = bitset(a, 8-k, 0);
   b(1,k+1) = a;
end
b
```

```
a =

  255


b =

   127    63    31    15     7     3     1     0
```

Set Multiple Bits

Set multiple bits to different values

```
bits = 2:6;
val = [1 0 0 1 1];
intout = bitset(0,bits,val,'int8')
```

```
intout =

      2     0     0    16    32
```

See Also

bitand | bitcmp | bitget | bitor | bitshift | bitxor | intmax

Purpose

Shift bits specified number of places

Compatibility

`bitshift(A,k,N)` will not accept the argument `N` in a future release. Use `bitshift(A,k)` or `bitcmp(A,k,assumedtype)` instead.

`bitshift` no longer interprets double values as 53-bit unsigned integers by default, but as `unit64` values.

Syntax

```
intout = bitshift(A,k)
intout = bitshift(A,k,assumedtype)

intout = bitshift(A,k,N)
```

Description

`intout = bitshift(A,k)` returns `A` shifted to the left by `k` bits, equivalent to multiplying by 2^k. Negative values of `k` correspond to shifting bits right or dividing by $2^{|k|}$ and rounding to the nearest integer towards negative infinite. Any overflow bits are truncated.

- If `A` is an array of signed integers, then `bitshift` returns the arithmetic shift results, preserving the signed bit when `k` is negative, and not preserving the signed bit when `k` is positive.
- If `k` is positive, MATLAB shifts the bits to the left and inserts `k` 0-bits on the right.
- If `k` is negative and `A` is nonnegative, then MATLAB shifts the bits to the right and inserts `|k|` 0-bits on the left.
- If `k` is negative and `A` is negative, then MATLAB shifts the bits to the right and inserts `|k|` 1-bits on the left.

`intout = bitshift(A,k,assumedtype)` assumes `A` is of type `assumedtype`.

`intout = bitshift(A,k,N)` truncates any bits that overflow `N` bits.

Input Arguments

A - Input value

signed integer array | unsigned integer array | double array

Input value, specified as a signed integer array, unsigned integer array, or double array.

- If `A` is a double array, and `assumedtype` is not specified, then MATLAB treats `A` as an unsigned 64-bit integer.
- If `assumedtype` is specified, then all elements in `A` must have integer values within the range of `assumedtype`.

Data Types

`double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

k - Number of switched bits

integer | integer array

Number of switched bits, specified as an integer or integer array.

Data Types

`double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

assumedtype - Assumed data type of A

`'uint64'` | `'uint32'` | `'uint16'` | `'uint8'` | `'int64'` | `'int32'` | `'int16'` | `'int8'`

Assumed data type of `A`, specified as `'uint64'`, `'uint32'`, `'uint16'`, `'uint8'`, `'int64'`, `'int32'`, `'int16'`, or `'int8'`.

- If `A` is an integer type array, then `assumedtype` must specify that same integer type.
- If `A` is a double array, then `assumedtype` can specify any valid integer type.

Data Types

`char`

N - Number of bits kept

nonnegative integer | nonnegative integer array

Number of bits kept, specified as a nonnegative integer or integer array. N must be less than or equal to the number of bits in the unsigned integer class of A.

Data Types

double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Output Arguments

intout - Shifted value

signed integer array | unsigned integer array | double array

Shifted value, returned as signed integer array, unsigned integer array, or double array. intout is the same size and type as A.

Examples

Shifted 8-bit Integer

Repeatedly shift the bits of an unsigned 8-bit value to the left until all the nonzero bits overflow.

```
a = intmax('uint8');
s1 = 'Initial uint8 value %5d is %08s in binary\n';
s2 = 'Shifted uint8 value %5d is %08s in binary\n';
fprintf(s1,a,dec2bin(a))
 for i = 1:8
    a = bitshift(a,1);
    fprintf(s2,a,dec2bin(a))
 end
```

```
Initial uint8 value   255 is 11111111 in binary
Shifted uint8 value   254 is 11111110 in binary
Shifted uint8 value   252 is 11111100 in binary
Shifted uint8 value   248 is 11111000 in binary
Shifted uint8 value   240 is 11110000 in binary
Shifted uint8 value   224 is 11100000 in binary
Shifted uint8 value   192 is 11000000 in binary
Shifted uint8 value   128 is 10000000 in binary
```

```
Shifted uint8 value     0 is 00000000 in binary
```

Different Results for Different Integer Types

Find the shift for a number using different assumed integer types.

```
uintout = bitshift(6,5:7,'uint8')
intout = bitshift(6,5:7,'int8')
```

```
uintout =

   192   128     0


intout =

   -64  -128     0
```

See Also

bitand | bitcmp | bitget | bitor | bitset | bitxor | intmax

Purpose

Bit-wise XOR

Syntax

```
intout = bitxor(integ1,integ2)
intout = bitxor(integ1,integ2,assumedtype)

objout = bitxor(netobj1,netobj2)
```

Description

`intout = bitxor(integ1,integ2)` returns the bit-wise XOR of `integ1` and `integ2`.

`intout = bitxor(integ1,integ2,assumedtype)` assumes that `integ1` and `integ2` are of `assumedtype`.

`objout = bitxor(netobj1,netobj2)` returns the bit-wise XOR of the .NET enumeration objects `netobj1` and `netobj2`.

Input Arguments

integ1, integ2 - Input values

signed integer arrays | unsigned integer arrays | double arrays

Input values, specified as signed integer arrays, unsigned integer arrays, or double arrays. `integ1` and `integ2` must be the same data type, or one must be a scalar double value.

- If `integ1` and `integ2` are double arrays, and `assumedtype` is not specified, then MATLAB treats `integ1` and `integ2` as unsigned 64-bit integers.
- If `assumedtype` is specified, then all elements in `integ1` and `integ2` must have integer values within the range of `assumedtype`.

Data Types

`double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

assumedtype - Assumed data type of integ1 and integ2

`'uint64' | 'uint32' | 'uint16' | 'uint8' | 'int64' | 'int32' | 'int16' | 'int8'`

Assumed data type of integ1 and integ2, specified as 'uint64', 'uint32', 'uint16', 'uint8', 'int64', 'int32', 'int16', or 'int8'.

- If integ1 and integ2 are double arrays, then assumedtype can specify any valid integer type, but defaults to 'uint64'.
- If integ1 and integ2 are integer type arrays, then assumedtype must specify that same integer type.

Data Types
char

netobj1, netobj2 - Input values
.NET enumeration objects

Input values, specified as .NET enumeration objects. You must be running a version of Windows to use .NET enumeration objects as input arguments.

Output Arguments

intout - Bit-wise XOR result
signed integer array | unsigned integer array | double array

Bit-wise XOR result, returned as a signed integer array, unsigned integer array, or double array. intout is the same data type and size as integ1 and integ2.

- If either integ1 or integ2 is a scalar double, and the other is a non-double integer type, intout is the non-double integer type.

objout - Bit-wise XOR result
.NET enumeration object

Bit-wise XOR result, returned as a .NET enumeration objects.

Examples

Truth Table

Create a truth table for the logical XOR operation.

```
A = uint8([0 1; 0 1]);
B = uint8([0 0; 1 1]);
TTable = bitxor(A, B)
```

```
TTable =

     0     1
     1     0
```

`bitxor` returns 0 if both bit-wise inputs are equal.

Negative Values

Explore how `bitxor` handles negative values.

MATLAB encodes negative integers using two's complement. Thus, the bit-wise XOR of -5 (11111010) and 6 (00000110) is -3 (11111100).

```
C = -5;
D = 6;
bitxor(C,D,'int8')
```

```
ans =

    -3
```

See Also

`bitand` | `bitcmp` | `bitget` | `bitor` | `bitshift` | `bitset` | `intmax`

Related Examples

- "Creating .NET Enumeration Bit Flags"

blanks

Purpose

Create string of blank characters

Syntax

```
blanks(n)
```

Description

`blanks(n)` is a string of `n` blanks.

Examples

`blanks` is useful with the `display` function. For example,

```
disp(['xxx' blanks(20) 'yyy'])
```

displays twenty blanks between the strings `'xxx'` and `'yyy'`.

`disp(blanks(n)')` moves the cursor down `n` lines.

See Also

`clc` | `format` | `home`

Purpose

Construct block diagonal matrix from input arguments

Syntax

```
out = blkdiag(a,b,c,d,...)
```

Description

`out = blkdiag(a,b,c,d,...)`, where `a`, `b`, `c`, `d`, `...` are matrices, outputs a block diagonal matrix of the form

$$\begin{bmatrix} a & 0 & 0 & 0 & 0 \\ 0 & b & 0 & 0 & 0 \\ 0 & 0 & c & 0 & 0 \\ 0 & 0 & 0 & d & 0 \\ 0 & 0 & 0 & 0 & \ldots \end{bmatrix}$$

The input matrices do not have to be square, nor do they have to be of equal size.

See Also

`diag` | `horzcat` | `vertcat`

box

Purpose

Axes border

Syntax

```
box on
box off
box
box(axes_handle,...)
```

Description

`box on` displays the boundary of the current axes.

`box off` does not display the boundary of the current axes.

`box` toggles the visible state of the current axes boundary.

`box(axes_handle,...)` uses the axes specified by `axes_handle` instead of the current axes.

Algorithms

The `box` function sets the axes `Box` property to `on` or `off`.

See Also

`axes` | `grid`

Purpose

Terminate execution of `for` or `while` loop

Syntax

```
break
```

Description

`break` terminates the execution of a `for` or `while` loop. Statements in the loop that appear after the `break` statement are not executed.

In nested loops, `break` exits only from the loop in which it occurs. Control passes to the statement that follows the `end` of that loop.

Tips

`break` is not defined outside a `for` or `while` loop. Use `return` in this context instead.

Examples

The example below shows a `while` loop that reads the contents of the file `fft.m` into a MATLAB character array. A `break` statement is used to exit the `while` loop when the first empty line is encountered. The resulting character array contains the command line help for the `fft` program.

```
fid = fopen('fft.m','r');
s = '';

while ~feof(fid)
   line = fgetl(fid);
   if isempty(line) || ~ischar(line), break, end
   s = sprintf('%s%s\n', s, line);
end
disp(s);

fclose(fid);
```

See Also

`for` | `while` | `end` | `continue` | `return`

brighten

Purpose Brighten or darken colormap

Syntax

```
brighten(beta)
brighten(h,beta)
newmap = brighten(beta)
newmap = brighten(cmap,beta)
```

Description `brighten(beta)` increases or decreases the color intensities in a colormap by replacing the current colormap with a brighter or darker colormap of essentially the same colors. The modified colormap is brighter if `0 < beta < 1` and darker if `-1 < beta < 0`. `brighten(beta)`, followed by `brighten(-beta)`, where `beta < 1`, restores the original map.

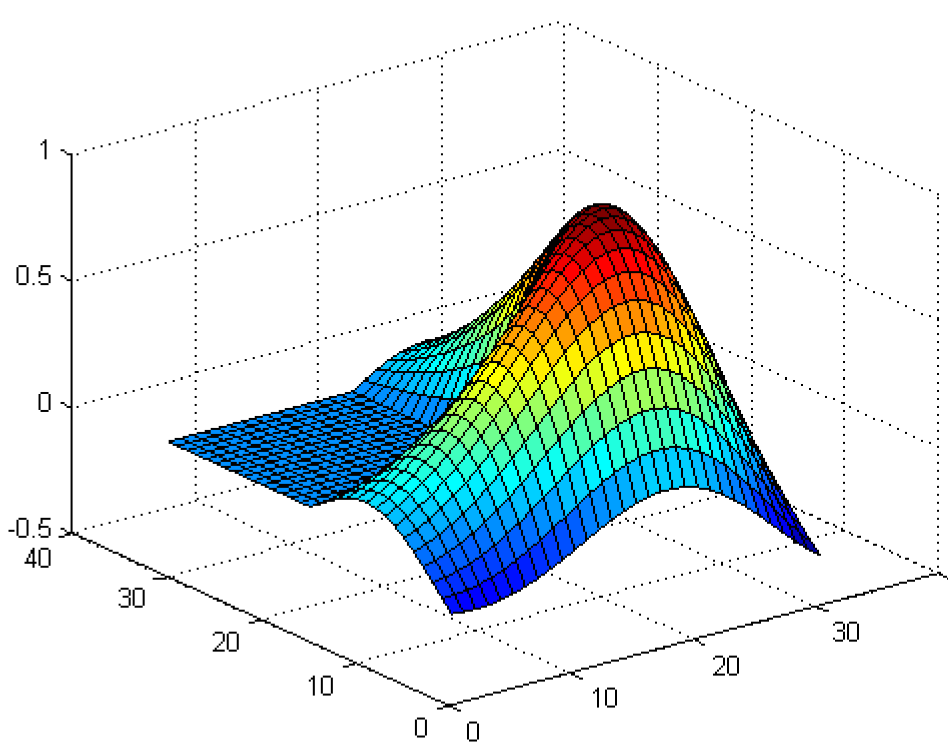
`brighten(h,beta)` brightens all objects that are children of the figure having the handle `h`.

`newmap = brighten(beta)` returns a brighter or darker version of the current colormap without changing the display.

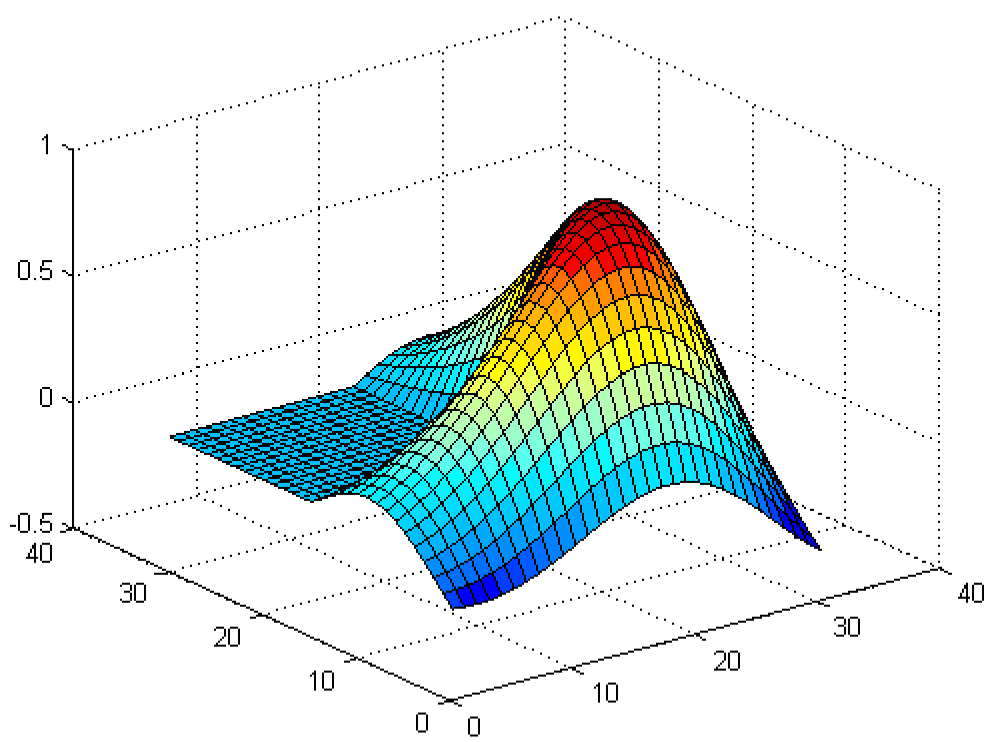
`newmap = brighten(cmap,beta)` returns a brighter or darker version of the colormap `cmap` without changing the display.

Examples Brighten the current colormap:

```
surf(membrane);
```




```
beta = .5;
brighten(beta);
```



Algorithms

`brighten` raises values in the colormap to the power of gamma, where gamma is

$$\gamma = \begin{cases} 1-\beta, & \beta > 0 \\ \dfrac{1}{1+\beta}, & \beta \le 0 \end{cases}$$

`brighten` has no effect on graphics objects defined with true color.

See Also

`colormap` | `rgbplot`

How To

- “Altering Colormaps”

brush

Purpose

Interactively mark, delete, modify, and save observations in graphs

Syntax

```
brush on
brush off
brush
brush color
brush(figure_handle,...)
brushobj = brush(figure_handle)
```

Description

Data brushing is a mode for interacting with graphs in figure windows in which you can click data points or drag a selection rectangle around data points to highlight observations in a color of your choice. Highlighting takes different forms for different types of graphs, and brushing marks persist—even in other interactive modes—until removed by deselecting them.

`brush on` turns on interactive data brushing mode.

`brush off` turns brushing mode off, leaving any brushed observations still highlighted.

`brush` by itself toggles the state of the data brushing tool.

`brush color` sets the current color used for brushing graphics to the specified `ColorSpec`. Changing brush color affects subsequent brushing, but does not change the color of observations already brushed or the brush tool's state.

`brush(figure_handle,...)` applies the function to the specified figure handle.

`brushobj = brush(figure_handle)` returns a *brush mode object* for that figure, useful for controlling and customizing the figure's brushing state. The following properties of such objects can be modified using `get` and `set`:

`Enable 'on' | {'off'}`	Specifies whether this figure mode is currently enabled on the figure.
`FigureHandle`	The associated figure handle. This property supports `get` only.
`Color`	Specifies the color to be used for brushing.

`brush` cannot return a brush mode object at the same time you are calling it to set a brushing option.

Tips

- “Types of Plots You Can Brush” on page 1-547
- “Plot Types You Cannot Brush” on page 1-549
- “Mode Exclusivity and Persistence” on page 1-550
- “How Data Linking Affects Data Brushing” on page 1-551
- “Mouse Gestures for Data Brushing” on page 1-552
- “Brush Mode Callbacks” on page 1-553

Types of Plots You Can Brush

Data brushing places lines and patches on plots to create highlighting, marking different types of graphs as follows (brushing marks are shown in red):

Graph Type	Brushing Annotation	Overlays?	Example
lineseries	Colored lines slightly wider than those in the lineseries with a marker distinct from those on the lineseries (filled circles if none) to identify brushed vertices. Only those line segments that connect brushed vertices are highlighted	Y	
scattergroup	Line with `LineStyle 'none'` and a marker with a color distinct from and slightly larger than the base scattergroup marker.	Y	
stemseries	The brushed stems and stem heads are shaded in the brushing color.	Y	
barseries	The interior of selected bars is filled in the brushing color.	N	
histogram	The bars to which brushed observations contribute are proportionately filled from the bottom up with the brushing color.	N	

Graph Type	Brushing Annotation	Overlays?	Example
areaseries	Patches filling the region between selected points and the *x*-axis in the brushing color.	N	
surfaceplot	Patches with edges slightly wider than the surfaceplot line width and with a marker distinct from that of the surfaceplot (**X** if none) to identify brushed vertices. Patches are plotted only when all four vertices that define them are brushed. The brushed observations are the set of marked vertices, not the patches.	N	

When using the linked plots feature, a graph can become brushed when you brush another graph that displays some of the same data, potentially brushing the same observations more than once. The overlaid brushing marks (whether lines or markers) are slightly wider than the brushing marks that they overlay; this makes multiply brushed observations visually distinct. The wider brushing marks are placed under the narrower ones, so that if they happen to have different colors, you can see all the colors. See the subsection "How Data Linking Affects Data Brushing" on page 1-551 for more information about brushing linked figures.

As the above table indicates, only lineseries, scatterseries, and stemseries brushing marks can be overlaid in this manner. Although you can brush them, you cannot overlay brushing marks on areaseries, barseries, histograms, or surfaceplots.

Plot Types You Cannot Brush

Currently, not all plot types enable data brushing. Graph functions that *do not* support brushing are:

- Line plots created with `line`
- Scatter plots created with `spy`
- Contour plots created with `contour`, `contourf`, or `contour3`
- Pie charts created with `pie` or `pie3`
- Radial graphs created with `polar`, `compass`, or `rose`
- Direction graphs created with `feather`, `quiver`, or `comet`
- Area and image plots created with `fill`, `image`, `imagesc`, or `pcolor`
- Bar graphs created with `pareto` or `errorbar`
- Functional plots created with `ezcontour` or `ezcontourf`
- 3-D plot types *other than* `plot3`, `stem3`, `scatter3`, `mesh`, `meshc`, `surf`, `surfl`, and `surfc`

You can use some of these functions to display base data that do not need to be brushable. For example, use `line` to plot mean *y*-values as horizontal lines that you do not need or want to brush.

Mode Exclusivity and Persistence

Data brushing mode is *exclusive*, like zoom, pan, data cursor, or plot edit mode. However, brush marks created in data brushing mode *persist* through all changes in mode. Brush marks that appear in other graphs while they are linked via `linkdata` also persist even when data linking is subsequently turned off. That is, severing connections to a graph's data sources does not remove brushing marks from it. The only ways to remove brushing marks are (in brushing mode):

- Brush an empty area in a brushed graph.
- Right-click and select **Clear all brushing** from the context menu.

Changing the brushing color for a figure does not recolor brushing marks on it until you brush it again. If you hold down the **Shift** key, all existing brush marks change to the new color. All brush marks that appear on linked plots in the same or different figure also change to the new color if the brushing action affects them. The behavior is the same whether you select a brushing color from the Brush Tool dropdown palette, set

it by calling `brush(colorspec)`, or by setting the `Color` property of a brush mode object (e.g., `set(brushobj,'Color',colorspec)`.

How Data Linking Affects Data Brushing

When you use the Data Linking tool or call the `linkdata` function, brushing marks that you make on one plot appear on other plots that depict the same variable you are brushing—if those plots are also linked. This happens even if the affected plot is not in Brushing mode. That is, brushing marks appear on a linked plot *in any mode* when you brush another plot linked to it via a common variable or brush that variable in the Variables editor. Be aware that the following conditions apply, however:

- The graph type must support data brushing (see "Types of Plots You Can Brush" on page 1-547 and "Plot Types You Cannot Brush" on page 1-549)
- The graphed variable must not be complex; if you can plot a complex variable you can brush it, but such graphs do not respond when you brush the complex variable in another linked plot. For more information about linking complex variables, see Example 3 in the `linkdata` reference page.
- Observations that you brush display in the same color in all linked graphs. The color is the brush color you have selected in the window you are interacting with, and can differ from the brushing colors selected in the other affected figures. When you brush linked plots, the brushing color is associated with the variable(s) you brush

The last bullet implies that brush marks on an unlinked graph can change color when data linking is turned on for that figure. Brushing marks can, in fact, vanish and be replaced by marks in the same or different color when the plot enters a linked state. In the linked state, brushing is tied to variables (data sources), not just the graphics. If different observations for the same variable on a linked figure are brushed, those variables override the brushed graphics on the newly linked plot. In other words, the newly linked graph loses all its previous brush marks when it "joins the club" of common data sources.

Mouse Gestures for Data Brushing

You can brush graphs in several ways. The basic operation is to drag the mouse to highlight all observations within the rectangle you define. The following table lists data brushing gestures and their effects.

Action	Gesture	Result
Select data using a region of interest	ROI mouse drag	Region of interest (ROI) rectangle (or rectangular prism for 3-D axes) appears during the gesture and all brushable observations within the rectangle are highlighted. All other brushing marks in the axes are removed. The ROI rectangle disappears when the mouse button is released.
Select a single point	Single left-click on a graphic object that supports data brushing	Produces an equivalent result to ROI rectangle, brushing where the rectangle encloses only the single vertex on the graphical object closest to the mouse. All other brushing annotations in the figure are removed.
Add a point to the selection or remove a highlighted one	Single left-click on a graphic object that supports data brushing, with the **Shift** key down	Equivalent brushing by dragging an ROI rectangle that encloses only the single vertex on the graphic object closest to the mouse. All other brushed regions in the figure remain brushed.
Select all data associated with a graphic object	Double left-click on a graphic object that supports data brushing	All vertices for the graphic object are brushed.

Action	Gesture	Result
Add to or subtract from region of interest	Click or ROI drag with the **Shift** or **Ctrl** keys down	Region of interest grows; all unbrushed vertices within the rectangle become brushed and all brushed observations in it become unbrushed. All brushed vertices outside the ROI remain brushed.
Copy brushed data to Editor, Command Window, Variables editor, or Workspace Browser	Drag brushed data to another window or to a program/icon on the system desktop	Equivalent to copying brushed data and pasting into other window or an existing/new variable.

Brush Mode Callbacks

You can program the following callbacks for brush mode operations.

- `ActionPreCallback <function_handle>` — Function to execute before brushing

  Use this callback to execute code when a brush operation begins. The function handle should reference a function with two implicit arguments:

  ```
  function myfunction(obj,event_obj)
  % obj         handle to the figure that has been clicked
  % event_obj   object containing struct of event data
  ```

  The event data `struct` has the following field:

`Axes`	The handle of the axes that is being brushed

- `ActionPostCallback <function_handle>` — Function to execute after brushing

  Use this callback to execute code when a brush operation ends. The function handle should reference a function with two implicit arguments:

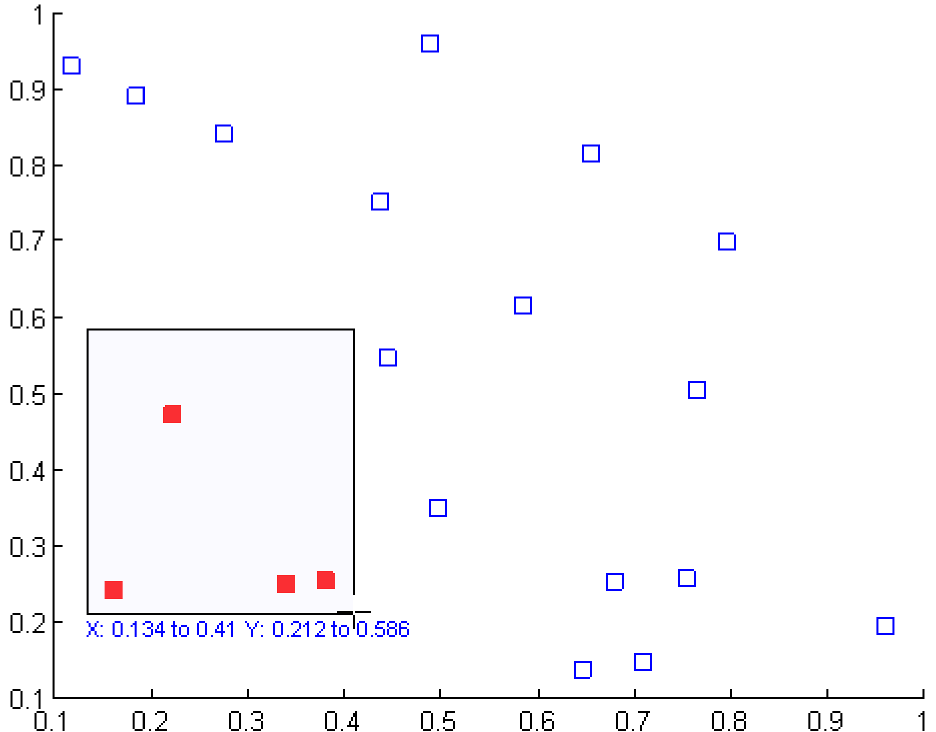
  ```
  function myfunction(obj,event_obj)
  % obj          handle to the figure that has been clicked
  % event_obj    object containing struct of event data (same as the
  %              event data of the 'ActionPreCallback' callback)
  ```

Examples

Example 1

On a scatterplot, drag out a rectangle to brush the graph:

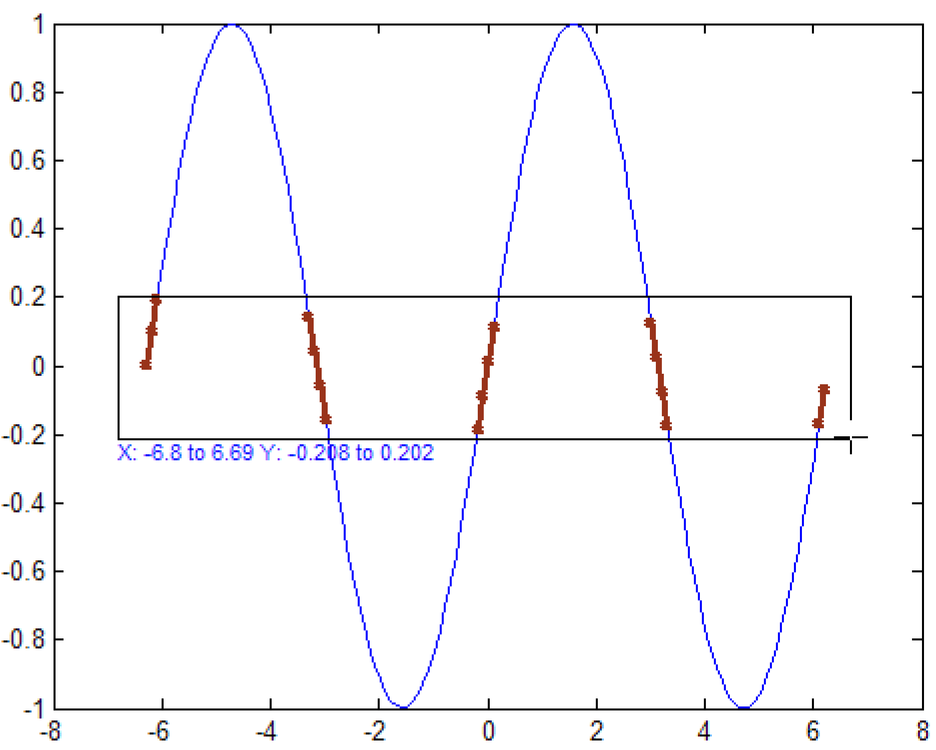
```
x = rand(20,1);
y = rand(20,1);
scatter(x,y,80,'s')
brush on
```



Example 2

Brush observations from -.2 to .2 on a lineseries plot in dark red:

```
x = [-2*pi:.1:2*pi];
y = sin(x);
plot(x,y);
h = brush;
set(h,'Color',[.6 .2 .1],'Enable','on');
```



See Also

`linkaxes` | `linkdata` | `pan` | `rotate3d` | `zoom`

How To

- "Marking Up Graphs with Data Brushing"

Purpose

Apply element-by-element binary operation to two arrays with singleton expansion enabled

Syntax

```
C = bsxfun(fun,A,B)
```

Description

`C = bsxfun(fun,A,B)` applies the element-by-element binary operation specified by the function handle `fun` to arrays `A` and `B`, with singleton expansion enabled. `fun` can be one of the following built-in functions:

`@plus`	Plus
`@minus`	Minus
`@times`	Array multiply
`@rdivide`	Right array divide
`@ldivide`	Left array divide
`@power`	Array power
`@max`	Binary maximum
`@min`	Binary minimum
`@rem`	Remainder after division
`@mod`	Modulus after division
`@atan2`	Four-quadrant inverse tangent; result in radians
`@atan2d`	Four-quadrant inverse tangent; result in degrees
`@hypot`	Square root of sum of squares
`@eq`	Equal
`@ne`	Not equal
`@lt`	Less than
`@le`	Less than or equal to
`@gt`	Greater than

@ge	Greater than or equal to
@and	Element-wise logical AND
@or	Element-wise logical OR
@xor	Logical exclusive OR

fun can also be a handle to any binary element-wise function not listed above. A binary element-wise function of the form C = fun(A,B) accepts arrays A and B of arbitrary but equal size and returns output of the same size. Each element in the output array C is the result of an operation on the corresponding elements of A and B only. fun must also support scalar expansion, such that if A or B is a scalar, C is the result of applying the scalar to every element in the other input array.

The corresponding dimensions of A and B must be equal to each other or equal to one. Whenever a dimension of A or B is singleton (equal to one), bsxfun virtually replicates the array along that dimension to match the other array. In the case where a dimension of A or B is singleton, and the corresponding dimension in the other array is zero, bsxfun virtually diminishes the singleton dimension to zero.

The size of the output array C is equal to:
max(size(A),size(B)).*(size(A)>0 & size(B)>0).

Examples

In this example, bsxfun is used to subtract the column means from the corresponding columns of matrix A.

```
A = magic(5);
A = bsxfun(@minus, A, mean(A))
A =

     4    11   -12    -5     2
    10    -8    -6     1     3
    -9    -7     0     7     9
    -3    -1     6     8   -10
    -2     5    12   -11    -4
```

See Also

`repmat` | `arrayfun`

builddocsearchdb

Purpose

Build searchable documentation database

Syntax

`builddocsearchdb(folder)`

Description

`builddocsearchdb(folder)` builds a searchable database from HTML files in the specified folder. The `builddocsearchdb` function creates a subfolder named `helpsearch` to contain the database files. The database enables MATLAB to search for content within the HTML files, as long as the version of MATLAB is the same as the version used to create the database.

Input Arguments

folder - Full path to folder with HTML files

string

Full path to a folder with HTML files, specified as a string.

To include a particular HTML document in the search database, the `builddocsearchdb` function requires that:

- The document has a title.
- The content is different from the title.

Example: `builddocsearchdb('c:\myfiles\html')`

Examples

Search Custom Help Files

Build a search database for custom help files.

MATLAB includes a set of sample files to demonstrate how to create a custom toolbox and supporting documentation. This sample toolbox is called the *Upslope Area Toolbox*. The `upslope` folder includes a file named `info.xml`, which is required to display custom documentation, and a subfolder named `html`, which contains HTML documentation and supporting files.

Copy the sample files to a temporary folder, and add the copied files to the path.

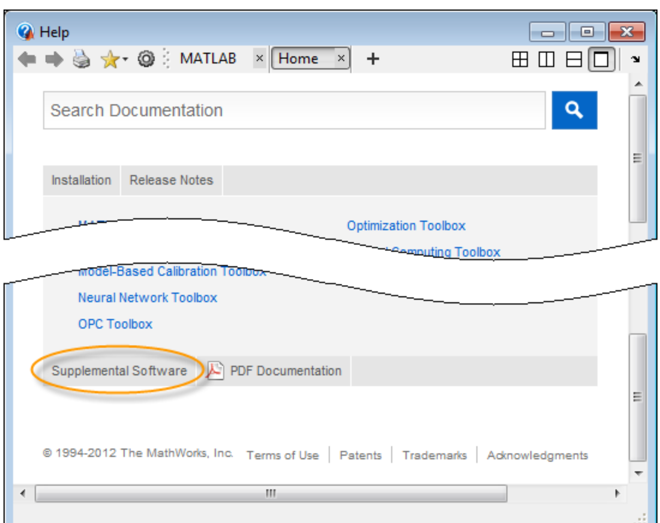
```
sample = fullfile(...
```

```
        matlabroot,'help','techdoc','matlab_env','examples','upslope
tmp = tempname;
mkdir(tmp);
copyfile(sample,tmp);
addpath(tmp);
```

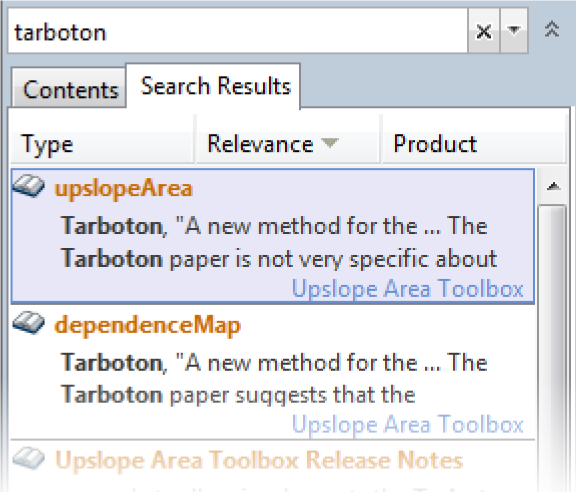
Create a search database.

```
folder = fullfile(tmp,'html');
builddocsearchdb(folder)
```

Open the Supplemental Software documentation browser. There is a link to this browser at the bottom of the documentation home page.



Search the supplemental documentation for the term *tarboton*, which appears in several of the example help files. The search returns several results from the Upslope Area Toolbox.



Remove the temporary example files.

```
rmpath(tmp)
rmdir(tmp,'s')
```

See Also

doc | help

Concepts

- “Display Custom Documentation”

Purpose

Execute built-in function from overloaded method

Syntax

```
builtin(function,x1,...,xn)
[y1,...,yn] = builtin(function,x1,...,xn)
```

Description

`builtin(function,x1,...,xn)` executes the built-in `function` with the input arguments `x1` through `xn`. Use `builtin` to execute the original built-in from within a method that overloads the function. To work properly, you must never overload `builtin`.

`[y1,...,yn] = builtin(function,x1,...,xn)` stores any output from `function` in `y1` through `yn`.

Input Arguments

function - Built-in function name
string

Valid built-in function name in the MATLAB path, specified as a string. `function` cannot be a function handle.

x1,...,xn - Valid input arguments for `function`
supported data types

Valid input arguments for `function`, specified by supported data types.

Examples

Run an Overloaded Function within a Class Definition

Execute the built-in functionality from within an overloaded method.

Create a simple class describing the speed of a particle and providing a `disp` method by pasting the following code into a file called `MyParticle.m`.

```
classdef MyParticle
    properties
        velocity;
    end
    methods
        function p = MyParticle(x,y,z)
```

```
            p.velocity.x = x;
            p.velocity.y = y;
            p.velocity.z = z;
        end
        function disp(p)
            builtin('disp',p) %call builtin
            if isscalar(p)
                disp('  Velocity')
                disp(['  x: ',num2str(p.velocity.x)])
                disp(['  y: ',num2str(p.velocity.y)])
                disp(['  z: ',num2str(p.velocity.z)])
            end
        end
    end
end
```

Create an instance MyParticle.

```
p = MyParticle(1,2,4)
```

```
p =

  MyParticle

  Properties:
    velocity: [1x1 struct]

  Methods

  Velocity
  x: 1
  y: 2
  z: 4
```

Definitions

built-in function

A built-in function is part of the MATLAB executable. MATLAB does not implement these functions in the MATLAB language. Although most built-in functions have a `.m` file associated with them, this file only supplies documentation for the function.

You can use the syntax `which` *`function`* to check whether a function is built-in.

See Also

`feval` | `which`

Purpose

Solve boundary value problems for ordinary differential equations

Syntax

```
sol = bvp4c(odefun,bcfun,solinit)
sol = bvp4c(odefun,bcfun,solinit,options)
solinit = bvpinit(x, yinit, params)
```

Arguments

`odefun`	A function handle that evaluates the differential equations $f(x,y)$. It can have the form
	`dydx = odefun(x,y)`
	`dydx = odefun(x,y,parameters)`
	where `x` is a scalar corresponding to x, and `y` is a column vector corresponding to y. `parameters` is a vector of unknown parameters. The output `dydx` is a column vector.

`bcfun`	A function handle that computes the residual in the boundary conditions. For two-point boundary value conditions of the form $bc(y(a),y(b))$, `bcfun` can have the form
	`res = bcfun(ya,yb)`
	`res = bcfun(ya,yb,parameters)`
	where `ya` and `yb` are column vectors corresponding to $y(a)$ and $y(b)$. `parameters` is a vector of unknown parameters. The output `res` is a column vector.
	See “Multipoint Boundary Value Problems” on page 1-569 for a description of `bcfun` for multipoint boundary value problems.

`solinit`	A structure containing the initial guess for a solution. You create `solinit` using the function `bvpinit`. `solinit` has the following fields.

	`x`	Ordered nodes of the initial mesh. Boundary conditions are imposed at a = `solinit.x(1)` and b = `solinit.x(end)`.
	`y`	Initial guess for the solution such that `solinit.y(:,i)` is a guess for the solution at the node `solinit.x(i)`.
	`parameters`	Optional. A vector that provides an initial guess for unknown parameters.
	The structure can have any name, but the fields must be named `x`, `y`, and `parameters`. You can form `solinit` with the helper function `bvpinit`. See `bvpinit` for details.	
`options`	Optional integration argument. A structure you create using the `bvpset` function. See `bvpset` for details.	

Description

`sol = bvp4c(odefun,bcfun,solinit)` integrates a system of ordinary differential equations of the form

$$y' = f(x,y)$$

on the interval [a,b] subject to two-point boundary value conditions $bc(y(a),y(b)) = 0$.

`odefun` and `bcfun` are function handles. See the `function_handle` reference page for more information.

“Parameterizing Functions” explains how to provide additional parameters to the function `odefun`, as well as the boundary condition function `bcfun`, if necessary.

`bvp4c` can also solve multipoint boundary value problems. See “Multipoint Boundary Value Problems” on page 1-569. You can use the function `bvpinit` to specify the boundary points, which are stored in the input argument `solinit`. See the reference page for `bvpinit` for more information.

The `bvp4c` solver can also find unknown parameters p for problems of the form

$$y' = f(x,y, p)$$

$$0 = bc(y(a),y(b),p)$$

where p corresponds to `parameters`. You provide `bvp4c` an initial guess for any unknown parameters in `solinit.parameters`. The `bvp4c` solver returns the final values of these unknown parameters in `sol.parameters`.

`bvp4c` produces a solution that is continuous on `[a,b]` and has a continuous first derivative there. Use the function `deval` and the output `sol` of `bvp4c` to evaluate the solution at specific points `xint` in the interval `[a,b]`.

```
sxint = deval(sol,xint)
```

The structure `sol` returned by `bvp4c` has the following fields:

`sol.x`	Mesh selected by `bvp4c`
`sol.y`	Approximation to $y(x)$ at the mesh points of `sol.x`
`sol.yp`	Approximation to $y'(x)$ at the mesh points of `sol.x`
`sol.parameters`	Values returned by `bvp4c` for the unknown parameters, if any
`sol.solver`	'bvp4c'
`sol.stats`	Computational cost statistics (also displayed when the `stats` option is set with `bvpset`).

The structure `sol` can have any name, and `bvp4c` creates the fields `x`, `y`, `yp`, `parameters`, and `solver`.

`sol = bvp4c(odefun,bcfun,solinit,options)` solves as above with default integration properties replaced by the values in `options`, a structure created with the `bvpset` function. See `bvpset` for details.

`solinit = bvpinit(x, yinit, params)` forms the initial guess `solinit` with the vector `params` of guesses for the unknown parameters.

Singular Boundary Value Problems

`bvp4c` solves a class of singular boundary value problems, including problems with unknown parameters `p`, of the form

$$y' = S \cdot y/x + F(x,y,p)$$

$$0 = bc(y(0),y(b),p)$$

The interval is required to be $[0, b]$ with $b > 0$. Often such problems arise when computing a smooth solution of ODEs that result from partial differential equations (PDEs) due to cylindrical or spherical symmetry. For singular problems, you specify the (constant) matrix `S` as the value of the `'SingularTerm'` option of `bvpset`, and `odefun` evaluates only $f(x,y,p)$. The boundary conditions must be consistent with the necessary condition $S \cdot y(0) = 0$ and the initial guess should satisfy this condition.

Multipoint Boundary Value Problems

`bvp4c` can solve multipoint boundary value problems where $a = a_0 < a_1 < a_2 < ...< a_n = b$ are boundary points in the interval $[a,b]$. The points $a_1,a_2,...,a_{n-1}$ represent interfaces that divide $[a,b]$ into regions. `bvp4c` enumerates the regions from left to right (from a to b), with indices starting from 1. In region k, $[a_{k-1},a_k]$, `bvp4c` evaluates the derivative as

```
yp = odefun(x,y,k)
```

In the boundary conditions function

```
bcfun(yleft,yright)
```

`yleft(:,k)` is the solution at the left boundary of $[a_{k-1},a_k]$. Similarly, `yright(:,k)` is the solution at the right boundary of region k. In particular,

```
yleft(:,1) = y(a)
```

and

```
yright(:,end) = y(b)
```

When you create an initial guess with

```
solinit = bvpinit(xinit,yinit),
```

use double entries in `xinit` for each interface point. See the reference page for `bvpinit` for more information.

If `yinit` is a function, `bvpinit` calls `y = yinit(x,k)` to get an initial guess for the solution at `x` in region `k`. In the solution structure `sol` returned by `bpv4c`, `sol.x` has double entries for each interface point. The corresponding columns of `sol.y` contain the left and right solution at the interface, respectively.

To see an example that solves a three-point boundary value problem, type `threebvp` at the MATLAB command prompt.

Note The `bvp5c` function is used exactly like `bvp4c`, with the exception of the meaning of error tolerances between the two solvers. If $S(x)$ approximates the solution $y(x)$, `bvp4c` controls the residual $|S'(x) - f(x,S(x))|$. This controls indirectly the true error $|y(x) - S(x)|$. `bvp5c` controls the true error directly. `bvp5c` is more efficient than `bvp4c` for small error tolerances.

Examples

Example 1

Boundary value problems can have multiple solutions and one purpose of the initial guess is to indicate which solution you want. The second-order differential equation

$$y'' + |y| = 0$$

has exactly two solutions that satisfy the boundary conditions $y(0) = 0$, $y(4) = -2$.

Prior to solving this problem with `bvp4c`, you must write the differential equation as a system of two first-order ODEs

$$y_1' = y_2$$

$$y_2' = -|y_1|.$$

Here $y_1 = y$ and $y_2 = y'$. This system has the required form

$$y' = f(x,y)$$

$$bc(y(a),y(b)) = 0$$

The function f and the boundary conditions bc are coded in MATLAB software as functions `twoode` and `twobc`.

```
function dydx = twoode(x,y)
  dydx = [ y(2)
           -abs(y(1))];

function res = twobc(ya,yb)
  res = [ ya(1)
          yb(1) + 2];
```

Form a guess structure consisting of an initial mesh of five equally spaced points in [0,4] and a guess of constant values

$$y_1(x) \equiv 0$$

and

$$y_2(x) \equiv 0$$

with the command

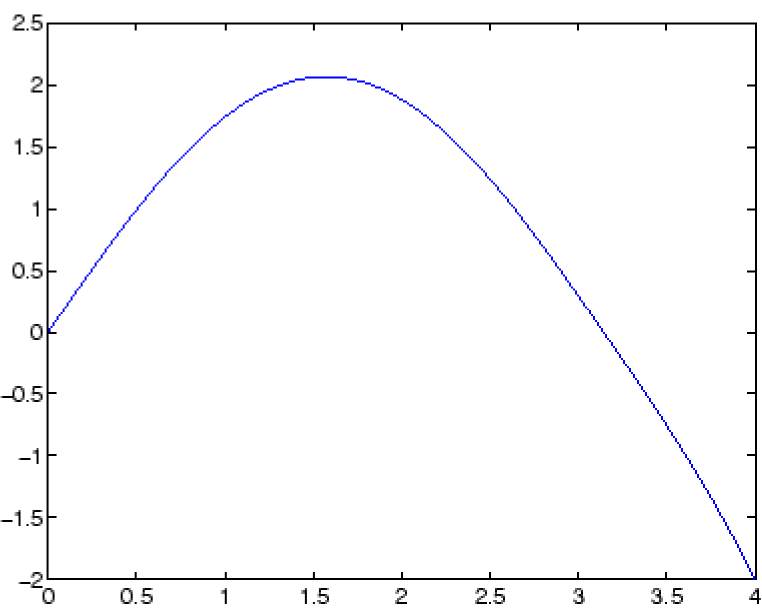
```
solinit = bvpinit(linspace(0,4,5),[1 0]);
```

Now solve the problem with

```
sol = bvp4c(@twoode,@twobc,solinit);
```

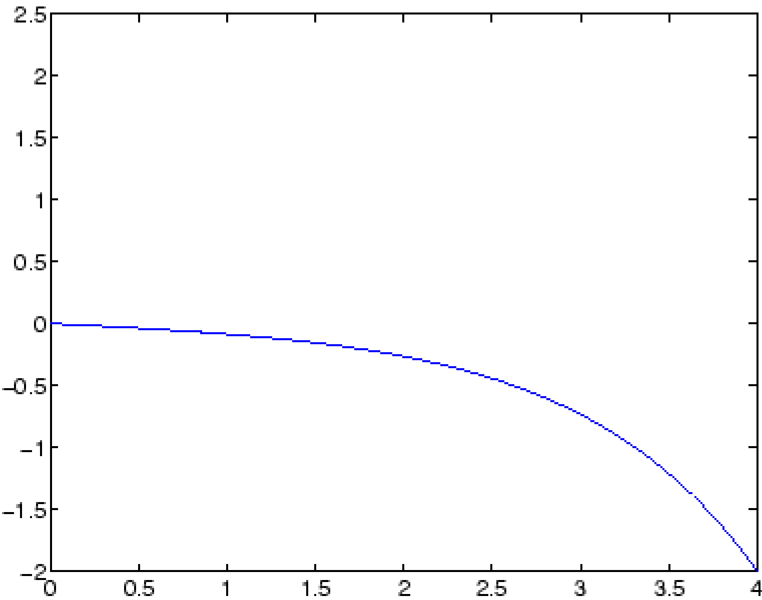
Evaluate the numerical solution at 100 equally spaced points and plot $y(x)$ with

```
x = linspace(0,4);
y = deval(sol,x);
plot(x,y(1,:));
```



You can obtain the other solution of this problem with the initial guess

```
solinit = bvpinit(linspace(0,4,5),[-1 0]);
```



Example 2

This boundary value problem involves an unknown parameter. The task is to compute the fourth ($q = 5$) eigenvalue λ of Mathieu's equation

$$y'' + (\lambda - 2q\cos 2x)y = 0.$$

Because the unknown parameter λ is present, this second-order differential equation is subject to *three* boundary conditions:

$$y'(0) = 0$$

$$y'(\pi) = 0$$

$$y(0) = 1$$

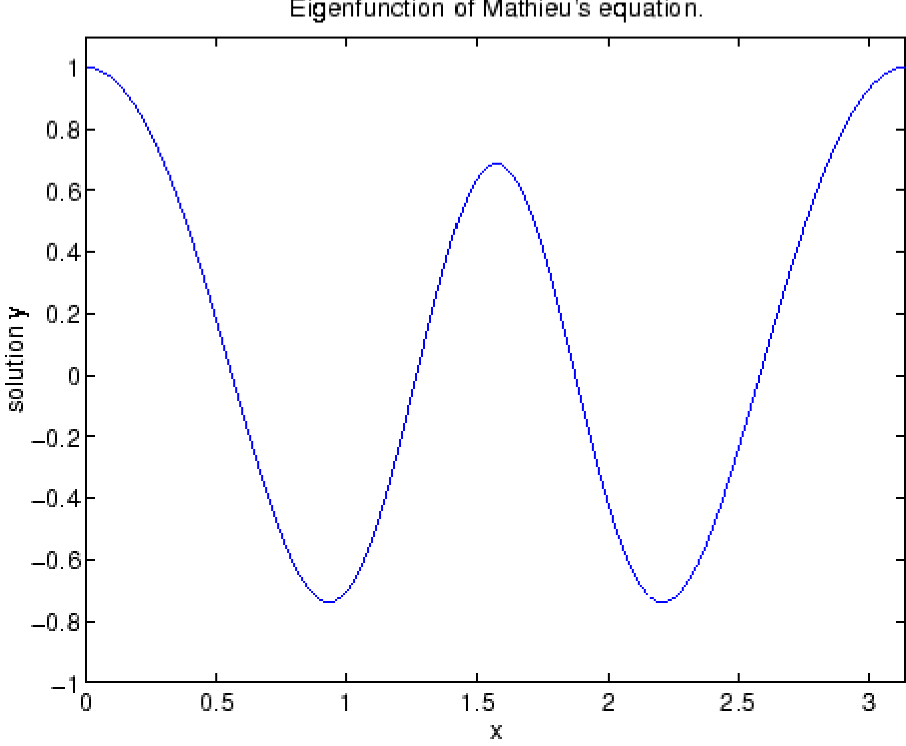
It is convenient to use local functions to place all the functions required by `bvp4c` in a single file.

```
function mat4bvp

lambda = 15;
solinit = bvpinit(linspace(0,pi,10),@mat4init,lambda);
sol = bvp4c(@mat4ode,@mat4bc,solinit);

fprintf('The fourth eigenvalue is approximately %7.3f.\n',...
        sol.parameters)

xint = linspace(0,pi);
Sxint = deval(sol,xint);
plot(xint,Sxint(1,:))
axis([0 pi -1 1.1])
title('Eigenfunction of Mathieu''s equation.')
xlabel('x')
ylabel('solution y')
% ------------------------------------------------------------
function dydx = mat4ode(x,y,lambda)
q = 5;
dydx = [  y(2)
         -(lambda - 2*q*cos(2*x))*y(1) ];
% ------------------------------------------------------------
function res = mat4bc(ya,yb,lambda)
res = [  ya(2)
         yb(2)
        ya(1)-1 ];
% ------------------------------------------------------------
function yinit = mat4init(x)
yinit = [  cos(4*x)
          -4*sin(4*x) ];
```

The differential equation (converted to a first-order system) and the boundary conditions are coded as local functions `mat4ode` and `mat4bc`, respectively. Because unknown parameters are present, these functions must accept three input arguments, even though some of the arguments are not used.

The guess structure `solinit` is formed with `bvpinit`. An initial guess for the solution is supplied in the form of a function `mat4init`. We chose $y = \cos 4x$ because it satisfies the boundary conditions and has the correct qualitative behavior (the correct number of sign changes). In the call to `bvpinit`, the third argument (`lambda = 15`) provides an initial guess for the unknown parameter λ.

After the problem is solved with `bvp4c`, the field `sol.parameters` returns the value $\lambda = 17.097$, and the plot shows the eigenfunction associated with this eigenvalue.



Algorithms

`bvp4c` is a finite difference code that implements the three-stage Lobatto IIIa formula. This is a collocation formula and the collocation

polynomial provides a C^1-continuous solution that is fourth-order accurate uniformly in `[a,b]`. Mesh selection and error control are based on the residual of the continuous solution.

References

[1] Shampine, L.F., M.W. Reichelt, and J. Kierzenka, "Solving Boundary Value Problems for Ordinary Differential Equations in MATLAB with bvp4c," available at `http://www.mathworks.com/bvp_tutorial`

See Also

`@` | `bvp5c` | `bvpget` | `bvpinit` | `bvpset` | `bvpxtend` | `deval`

Purpose

Solve boundary value problems for ordinary differential equations

Syntax

```
sol = bvp5c(odefun,bcfun,solinit)
sol = bvp5c(odefun,bcfun,solinit,options)
solinit = bvpinit(x, yinit, params)
```

Arguments

`odefun`	A function handle that evaluates the differential equations $f(x,y)$. It can have the form
	
	`dydx = odefun(x,y)`
	`dydx = odefun(x,y,parameters)`
	
	where `x` is a scalar corresponding to x, and `y` is a column vector corresponding to y. `parameters` is a vector of unknown parameters. The output `dydx` is a column vector.	

`bcfun`	A function handle that computes the residual in the boundary conditions. For two-point boundary value conditions of the form $bc(y(a),y(b))$, `bcfun` can have the form
	
	`res = bcfun(ya,yb)`
	`res = bcfun(ya,yb,parameters)`
	
	where `ya` and `yb` are column vectors corresponding to $y(a)$ and $y(b)$. `parameters` is a vector of unknown parameters. The output `res` is a column vector.	

`solinit`	A structure containing the initial guess for a solution. You create `solinit` using the function `bvpinit`. `solinit` has the following fields.	
	`x`	Ordered nodes of the initial mesh. Boundary conditions are imposed at a = `solinit.x(1)` and b = `solinit.x(end)`.

	y	Initial guess for the solution such that `solinit.y(:,i)` is a guess for the solution at the node `solinit.x(i)`.
	parameters	Optional. A vector that provides an initial guess for unknown parameters.
	The structure can have any name, but the fields must be named `x`, `y`, and `parameters`. You can form `solinit` with the helper function `bvpinit`. See `bvpinit` for details.	
options	Optional integration argument. A structure you create using the `bvpset` function. See `bvpset` for details.	

Description

`sol = bvp5c(odefun,bcfun,solinit)` integrates a system of ordinary differential equations of the form

$$y' = f(x,y)$$

on the interval [a,b] subject to two-point boundary value conditions

$$bc(y(a),y(b)) = 0$$

`odefun` and `bcfun` are function handles. See the `function_handle` reference page for more information.

"Parameterizing Functions" explains how to provide additional parameters to the function `odefun`, as well as the boundary condition function `bcfun`, if necessary. You can use the function `bvpinit` to specify the boundary points, which are stored in the input argument `solinit`.

The `bvp5c` solver can also find unknown parameters p for problems of the form

$$y' = f(x,y,p)$$

$$0 = bc(y(a),y(b),p)$$

where p corresponds to `parameters`. You provide `bvp5c` an initial guess for any unknown parameters in `solinit.parameters`. The `bvp5c` solver returns the final values of these unknown parameters in `sol.parameters`.

`bvp5c` produces a solution that is continuous on `[a,b]` and has a continuous first derivative there. Use the function `deval` and the output `sol` of `bvp5c` to evaluate the solution at specific points `xint` in the interval `[a,b]`.

```
sxint = deval(sol,xint)
```

The structure `sol` returned by `bvp5c` has the following fields:

`sol.x`	Mesh selected by `bvp5c`
`sol.y`	Approximation to $y(x)$ at the mesh points of `sol.x`
`sol.parameters`	Values returned by `bvp5c` for the unknown parameters, if any
`sol.solver`	`'bvp5c'`
`sol.stats`	Computational cost statistics (also displayed when the `stats` option is set with `bvpset`).

The structure `sol` can have any name, and `bvp5c` creates the fields `x`, `y`, `parameters`, and `solver`.

`sol = bvp5c(odefun,bcfun,solinit,options)` solves as above with default integration properties replaced by the values in `options`, a structure created with the `bvpset` function. See `bvpset` for details.

`solinit = bvpinit(x, yinit, params)` forms the initial guess `solinit` with the vector `params` of guesses for the unknown parameters.

Singular Boundary Value Problems

`bvp5c` solves a class of singular boundary value problems, including problems with unknown parameters `p`, of the form

$$y' = S \cdot y/x + f(x,y,p)$$

$$0 = bc(y(0),y(b),p)$$

The interval is required to be [0, b] with b > 0. Often such problems arise when computing a smooth solution of ODEs that result from partial differential equations (PDEs) due to cylindrical or spherical symmetry. For singular problems, you specify the (constant) matrix `S` as the value of the `'SingularTerm'` option of `bvpset`, and `odefun` evaluates only $f(x,y,p)$. The boundary conditions must be consistent with the necessary condition $S \cdot y(0) = 0$ and the initial guess should satisfy this condition.

Multipoint Boundary Value Problems

`bvp5c` can solve multipoint boundary value problems where $a = a_0 < a_1 < a_2 < ... < a_n = b$ are boundary points in the interval $[a,b]$. The points $a_1,a_2, ... ,a_{n-1}$ represent interfaces that divide $[a,b]$ into regions. `bvp5c` enumerates the regions from left to right (from a to b), with indices starting from 1. In region k, $[a_{k-1},a_k]$, `bvp5c` evaluates the derivative as

```
yp = odefun(x,y,k)
```

In the boundary conditions function

```
bcfun(yleft,yright)
```

yleft(:, k) is the solution at the left boundary of $[a_{k-1},a_k]$. Similarly, `yright(:,k)` is the solution at the right boundary of region k. In particular,

```
yleft(:,1) = y(a)
```

and

```
yright(:,end) = y(b)
```

When you create an initial guess with

```
solinit = bvpinit(xinit,yinit),
```

use double entries in `xinit` for each interface point. See the reference page for `bvpinit` for more information.

If `yinit` is a function, `bvpinit` calls `y = yinit(x, k)` to get an initial guess for the solution at `x` in region `k`. In the solution structure `sol` returned by `bvp5c`, `sol.x` has double entries for each interface point. The corresponding columns of `sol.y` contain the left and right solution at the interface, respectively.

To see an example of that solves a three-point boundary value problem, type `threebvp` at the MATLAB command prompt.

Algorithms

`bvp5c` is a finite difference code that implements the four-stage Lobatto IIIa formula. This is a collocation formula and the collocation polynomial provides a C^1-continuous solution that is fifth-order accurate uniformly in `[a,b]`. The formula is implemented as an implicit Runge-Kutta formula. `bvp5c` solves the algebraic equations directly; `bvp4c` uses analytical condensation. `bvp4c` handles unknown parameters directly; while `bvp5c` augments the system with trivial differential equations for unknown parameters.

References

[1] Shampine, L.F., M.W. Reichelt, and J. Kierzenka "Solving Boundary Value Problems for Ordinary Differential Equations in MATLAB with bvp4c" `http://www.mathworks.com/bvp_tutorial`. Note that this tutorial uses the bvp4c function, however in most cases the solvers can be used interchangeably.

See Also

`@` | `bvp4c` | `bvpget` | `bvpinit` | `bvpset` | `bvpxtend` | `deval`

bvpget

Purpose

Extract properties from options structure created with bvpset

Syntax

```
val = bvpget(options,'name')
val = bvpget(options,'name',default)
```

Description

`val = bvpget(options,'name')` extracts the value of the named property from the structure `options`, returning an empty matrix if the property value is not specified in `options`. It is sufficient to type only the leading characters that uniquely identify the property. Case is ignored for property names. `[]` is a valid `options` argument.

`val = bvpget(options,'name',default)` extracts the named property as above, but returns `val = default` if the named property is not specified in `options`. For example,

```
val = bvpget(options,'RelTol',1e-4);
```

returns `val = 1e-4` if the `RelTol` is not specified in `options`.

See Also

`bvp4c` | `bvp5c` | `bvpinit` | `bvpset` | `deval`

Purpose

Form initial guess for BVP solvers

Syntax

```
solinit = bvpinit(x,yinit)
solinit = bvpinit(x,yinit,parameters)
solinit = bvpinit(sol,[anew bnew])
solinit = bvpinit(sol,[anew bnew],parameters)
```

Description

`solinit = bvpinit(x,yinit)` forms the initial guess for a boundary value problem solver.

`x` is a vector that specifies an initial mesh. If you want to solve the BVP on [a,b], then specify `x(1)` as a and `x(end)` as b. The solver adapts this mesh to the solution, so a guess like `xb=nlinspace(a,b,10)` often suffices. However, in difficult cases, you should place mesh points where the solution changes rapidly. The entries of `x` must be in

- Increasing order if $a<b$
- Decreasing order if $a>b$

For two-point boundary value problems, the entries of `x` must be distinct. That is, if $a<b$, the entries must satisfy `x(1)` < `x(2)` < ... < `x(end)`. If $a>b$, the entries must satisfy `x(1)` > `x(2)` > ... > `x(end)`

For multipoint boundary value problem, you can specify the points in [a,b] at which the boundary conditions apply, other than the endpoints a and b, by repeating their entries in `x`. For example, if you set

```
x = [0, 0.5, 1, 1,  1.5, 2];
```

the boundary conditions apply at three points: the endpoints `0` and `2`, and the repeated entry 1. In general, repeated entries represent boundary points between regions in [a,b]. In the preceding example, the repeated entry 1 divides the interval `[0,2]` into two regions: `[0,1]` and `[1,2]`.

`yinit` is a guess for the solution. It can be either a vector, or a function:

- Vector – For each component of the solution, `bvpinit` replicates the corresponding element of the vector as a constant guess across

all mesh points. That is, `yinit(i)` is a constant guess for the `i`th component `yinit(i,:)` of the solution at all the mesh points in `x`.

- Function – For a given mesh point, the guess function must return a vector whose elements are guesses for the corresponding components of the solution. The function must be of the form

  ```
  y = guess(x)
  ```

  where `x` is a mesh point and `y` is a vector whose length is the same as the number of components in the solution. For example, if the guess function is a function, `bvpinit` calls

  ```
  y(:,j) = guess(x(j))
  ```

  at each mesh point.

  For multipoint boundary value problems, the guess function must be of the form

  ```
  y = guess(x, k)
  ```

  where `y` an initial guess for the solution at `x` in region `k`. The function must accept the input argument `k`, which is provided for flexibility in writing the guess function. However, the function is not required to use `k`.

`solinit = bvpinit(x,yinit,parameters)` indicates that the boundary value problem involves unknown parameters. Use the vector `parameters` to provide a guess for all unknown parameters.

`solinit` is a structure with the following fields. The structure can have any name, but the fields must be named `x`, `y`, and `parameters`.

`x`	Ordered nodes of the initial mesh.
`y`	Initial guess for the solution with `solinit.y(:,i)` a guess for the solution at the node `solinit.x(i)`.
`parameters`	Optional. A vector that provides an initial guess for unknown parameters.

`solinit = bvpinit(sol,[anew bnew])` forms an initial guess on the interval `[anew bnew]` from a solution `sol` on an interval [a,b]. The new interval must be larger than the previous one, so either `anew <=` $a < b$ `<= bnew` or `anew >=` $a > b$ `>= bnew`. The solution `sol` is extrapolated to the new interval. If `sol` contains `parameters`, they are copied to `solinit`.

`solinit = bvpinit(sol,[anew bnew],parameters)` forms `solinit` as described above, but uses `parameters` as a guess for unknown parameters in `solinit`.

See Also

`@` | `bvp4c` | `bvp5c` | `bvpget` | `bvpset` | `bvpxtend` | `deval`

bvpset

Purpose

Create or alter options structure of boundary value problem

Syntax

```
options = bvpset('name1',value1,'name2',value2,...)
options = bvpset(oldopts,'name1',value1,...)
options = bvpset(oldopts,newopts)
bvpset
```

Description

`options = bvpset('name1',value1,'name2',value2,...)` creates a structure `options` that you can supply to the boundary value problem solver `bvp4c`, in which the named properties have the specified values. Any unspecified properties retain their default values. For all properties, it is sufficient to type only the leading characters that uniquely identify the property. `bvpset` ignores case for property names.

`options = bvpset(oldopts,'name1',value1,...)` alters an existing options structure `oldopts`. This overwrites any values in `oldopts` that are specified using name/value pairs and returns the modified structure as the output argument.

`options = bvpset(oldopts,newopts)` combines an existing options structure `oldopts` with a new options structure `newopts`. Any values set in `newopts` overwrite the corresponding values in `oldopts`.

`bvpset` with no input arguments displays all property names and their possible values, indicating defaults with braces `{}`.

You can use the function `bvpget` to query the `options` structure for the value of a specific property.

BVP Properties

`bvpset` enables you to specify properties for the boundary value problem solver `bvp4c`. There are several categories of properties that you can set:

- "Error Tolerance Properties" on page 1-587
- "Vectorization" on page 1-588
- "Analytical Partial Derivatives" on page 1-589
- "Singular BVPs" on page 1-590
- "Mesh Size Property" on page 1-591

- "Solution Statistic Property" on page 1-592

Error Tolerance Properties

Because `bvp4c` uses a collocation formula, the numerical solution is based on a mesh of points at which the collocation equations are satisfied. Mesh selection and error control are based on the residual of this solution, such that the computed solution $S(x)$ is the exact solution of a perturbed problem $S'(x) = f(x,S(x)) + res(x)$. On each subinterval of the mesh, a norm of the residual in the `i`th component of the solution, `res(i)`, is estimated and is required to be less than or equal to a tolerance. This tolerance is a function of the relative and absolute tolerances, `RelTol` and `AbsTol`, defined by the user.

$$\left\| \left(\mathrm{res}(i) / \max\left(\mathrm{abs}\left(f(i) \right), \mathrm{AbsTol}(i) / \mathrm{RelTol} \right) \right) \right\| \le \mathrm{RelTol}$$

The following table describes the error tolerance properties.

BVP Error Tolerance Properties

Property	Value	Description
`RelTol`	Positive scalar {`1e-3`}	A relative error tolerance that applies to all components of the residual vector. It is a measure of the residual relative to the size of $f(x,y)$. The default, `1e-3`, corresponds to 0.1% accuracy. The computed solution $S(x)$ is the exact solution of $S'(x) = F(x,S(x)) + \mathrm{res}(x)$. On each subinterval of the mesh, the residual $\mathrm{res}(x)$ satisfies $\left\| \left(\mathrm{res}(i) / \max\left(\mathrm{abs}\left(F(i) \right), \mathrm{AbsTol}(i) / \mathrm{RelTol} \right) \right) \right\| \le \mathrm{RelTol}$
`AbsTol`	Positive scalar or vector {`1e-6`}	Absolute error tolerances that apply to the corresponding components of the residual vector. `AbsTol(i)` is a threshold below which the values of the corresponding components are unimportant. If a scalar value is specified, it applies to all components.

Vectorization

The following table describes the BVP vectorization property. Vectorization of the ODE function used by bvp4c differs from the vectorization used by the ODE solvers:

- For bvp4c, the ODE function must be vectorized with respect to the first argument as well as the second one, so that F([x1 x2 ...],[y1 y2 ...]) returns [F(x1,y1) F(x2,y2)...].
- bvp4c benefits from vectorization even when analytical Jacobians are provided. For stiff ODE solvers, vectorization is ignored when analytical Jacobians are used.

Vectorization Properties

Property	Value	Description
Vectorized	on | {off}	Set on to inform bvp4c that you have coded the ODE function F so that F([x1 x2 ...],[y1 y2 ...]) returns [F(x1,y1) F(x2,y2) ...]. That is, your ODE function can pass to the solver a whole array of column vectors at once. This enables the solver to reduce the number of function evaluations and may significantly reduce solution time. With the MATLAB array notation, it is typically an easy matter to vectorize an ODE function. In the shockbvp example shown previously, the shockODE function has been vectorized using colon notation into the subscripts and by using the array multiplication (.*) operator. `function dydx = shockODE(x,y,e)`

Vectorization Properties (Continued)

Property	Value	Description

		`pix = pi*x;`
		`dydx = [ y(2,:)...`
		`-x/e.*y(2,:)-pi^2*cos(pix)-`
		`pix/e.*sin(pix)];`

Analytical Partial Derivatives

By default, the `bvp4c` solver approximates all partial derivatives with finite differences. `bvp4c` can be more efficient if you provide analytical partial derivatives $\partial f/\partial y$ of the differential equations, and analytical partial derivatives, $\partial bc/\partial ya$ and $\partial bc/\partial yb$, of the boundary conditions. If the problem involves unknown parameters, you must also provide partial derivatives, $\partial f/\partial p$ and $\partial bc/\partial p$, with respect to the parameters.

The following table describes the analytical partial derivatives properties.

BVP Analytical Partial Derivative Properties

Property	Value	Description
`FJacobian`	Function handle	A function handle that computes the analytical partial derivatives of $f(x,y)$. When solving $y' = f(x,y)$, set this property to `@fjac` if `dfdy = fjac(x,y)` evaluates the Jacobian $\partial f/\partial y$. If the problem involves unknown parameters p, `[dfdy,dfdp] = fjac(x,y,p)` must also return the partial derivative $\partial f/\partial p$. For problems with constant partial derivatives, set

BVP Analytical Partial Derivative Properties (Continued)

Property	Value	Description
		this property to the value of `dfdy` or to a cell array `{dfdy,dfdp}`.
`BCJacobian`	Function handle	A function handle that computes the analytical partial derivatives of *bc*(*ya*,*yb*). For boundary conditions *bc*(*ya*,*yb*), set this property to `@bcjac` if `[dbcdya,dbcdyb] = bcjac(ya,yb)` evaluates the partial derivatives $\partial bc/\partial ya$, and $\partial bc/\partial yb$. If the problem involves unknown parameters *p*, `[dbcdya,dbcdyb,dbcdp] = bcjac(ya,yb,p)` must also return the partial derivative $\partial bc/\partial p$. For problems with constant partial derivatives, set this property to a cell array `{dbcdya,dbcdyb}` or `{dbcdya,dbcdyb,dbcdp}`.

Singular BVPs

`bvp4c` can solve singular problems of the form

$$y' = S\frac{y}{x} + f(x, y, p)$$

posed on the interval [0,*b*] where $b > 0$. For such problems, specify the constant matrix *S* as the value of `SingularTerm`. For equations of this form, `odefun` evaluates only the $f(x,y,p)$ term, where *p* represents unknown parameters, if any.

Singular BVP Property

Property	Value	Description
SingularTerm	Constant matrix	Singular term of singular BVPs. Set to the constant matrix S for equations of the form $$y' = S\frac{y}{x} + f(x, y, p)$$ posed on the interval [0,b] where $b > 0$.

Mesh Size Property

bvp4c solves a system of algebraic equations to determine the numerical solution to a BVP at each of the mesh points. The size of the algebraic system depends on the number of differential equations (n) and the number of mesh points in the current mesh (N). When the allowed number of mesh points is exhausted, the computation stops, bvp4c displays a warning message and returns the solution it found so far. This solution does not satisfy the error tolerance, but it may provide an excellent initial guess for computations restarted with relaxed error tolerances or an increased value of NMax.

The following table describes the mesh size property.

BVP Mesh Size Property

Property	Value	Description
NMax	positive integer {floor(1000/n)}	Maximum number of mesh points allowed when solving the BVP, where n is the number of differential equations in the problem. The default value of NMax limits the size of the algebraic system to about 1000 equations. For systems of a few differential equations, the default value of NMax should be sufficient to obtain an accurate solution.

Solution Statistic Property

The Stats property lets you view solution statistics.

The following table describes the solution statistics property.

BVP Solution Statistic Property

Property	Value	Description
Stats	on | {off}	Specifies whether statistics about the computations are displayed. If the stats property is on, after solving the problem, bvp4c displays: • The number of points in the mesh • The maximum residual of the solution • The number of times it called the differential equation function odefun to evaluate $f(x,y)$

BVP Solution Statistic Property (Continued)

Property	Value	Description
		• The number of times it called the boundary condition function `bcfun` to evaluate $bc(y(a),y(b))$

Examples

To create an options structure that changes the relative error tolerance of `bvp4c` from the default value of `1e-3` to `1e-4`, enter

```
options = bvpset('RelTol',1e-4);
```

To recover the value of `'RelTol'` from `options`, enter

```
bvpget(options,'RelTol')

ans =

  1.0000e-004
```

See Also

`function_handle` | `bvp4c` | `bvp5c` | `bvpget` | `bvpinit` | `deval`

bvpxtend

Purpose

Form guess structure for extending boundary value solutions

Syntax

```
solinit = bvpxtend(sol,xnew,ynew)
solinit = bvpxtend(sol,xnew,extrap)
solinit = bvpxtend(sol,xnew)
solinit = bvpxtend(sol,xnew,ynew,pnew)
solinit = bvpxtend(sol,xnew,extrap,pnew)
```

Description

`solinit = bvpxtend(sol,xnew,ynew)` uses solution `sol` computed on `[a,b]` to form a solution guess for the interval extended to `xnew`. The extension point `xnew` must be outside the interval `[a,b]`, but on either side. The vector `ynew` provides an initial guess for the solution at `xnew`.

`solinit = bvpxtend(sol,xnew,extrap)` forms the guess at `xnew` by extrapolating the solution `sol`. `extrap` is a string that determines the extrapolation method. `extrap` has three possible values:

- `'constant'` — `ynew` is a value nearer to end point of solution in `sol`.
- `'linear'` — `ynew` is a value at `xnew` of linear interpolant to the value and slope at the nearer end point of solution in `sol`.
- `'solution'` — `ynew` is the value of (cubic) solution in `sol` at `xnew`.

The value of `extrap` is case-insensitive and only the leading, unique portion needs to be specified.

`solinit = bvpxtend(sol,xnew)` uses the extrapolating solution where `extrap` is `'constant'`. If there are unknown parameters, values present in `sol` are used as the initial guess for parameters in `solinit`.

`solinit = bvpxtend(sol,xnew,ynew,pnew)` specifies a different guess `pnew`. `pnew` can be used with extrapolation, using the syntax `solinit = bvpxtend(sol,xnew,extrap,pnew)`. To modify parameters without changing the interval, use `[]` as place holder for `xnew` and `ynew`.

See Also

`bvp4c` | `bvp5c` | `bvpinit`

Purpose

Calendar for specified month

Syntax

```
c = calendar
c = calendar(d)
c = calendar(y, m)
```

Description

`c = calendar` returns a 6-by-7 matrix containing a calendar for the current month. The calendar runs Sunday (first column) to Saturday.

`c = calendar(d)`, where `d` is a serial date number or a date string, returns a calendar for the specified month.

`c = calendar(y, m)`, where `y` and `m` are integers, returns a calendar for the specified month of the specified year.

Examples

The command

```
calendar(1957,10)
```

reveals that the Space Age began on a Friday (on October 4, 1957, when Sputnik 1 was launched).

```
                          Oct 1957
     S      M     Tu      W     Th      F      S
     0      0      1      2      3      4      5
     6      7      8      9     10     11     12
    13     14     15     16     17     18     19
    20     21     22     23     24     25     26
    27     28     29     30     31      0      0
     0      0      0      0      0      0      0
```

See Also

`datenum`

calllib

Purpose

Call function in shared library

Syntax

```
[x1,...,xN] = calllib(libname,funcname,arg1,...,argN)
```

Description

`[x1,...,xN] = calllib(libname,funcname,arg1,...,argN)` calls function, `funcname`, in library, `libname`, passing input arguments, `arg1,...,argN`, and returns output values obtained from `funcname` in `x1,...,xN`.

Input Arguments

libname - Name of shared library

string

Name of shared library, specified as a string. If you call `loadlibrary` using the `alias` option, then you must use the alias name for the `libname` argument.

Data Types

`char`

funcname - Name of function in library

string

Name of function in library, specified as a string.

Data Types

`char`

arg1,...,argN - Input arguments

any type

Input arguments, 1 through N, required by `funcname` (if any), specified by any type. The argument type is specified by the `funcname` argument list.

Output Arguments

x1,...,xN - Output arguments

any type

Output arguments, 1 through N, from `funcname` (if any), returned as any type. The argument type is specified by the `funcname` argument list.

Limitations

- Use with libraries that are loaded using the `loadlibrary` function.

Examples

Call addStructByRef Function

Load the library.

```
if ~libisloaded('shrlibsample')
   addpath(fullfile(matlabroot,'extern','examples','shrlib'))
   loadlibrary('shrlibsample')
end
```

Display function signature.

```
libfunctionsview shrlibsample
```

```
[double, c_structPtr] addStructByRef(c_structPtr)
```

The input argument is a pointer to a `c_struct` data type.

Create a MATLAB structure, `struct`:

```
struct.p1 = 4; struct.p2 = 7.3; struct.p3 = -290;
```

Call the function.

```
[res,st] = calllib('shrlibsample','addStructByRef',struct);
```

Display the results.

```
res
```

```
res =
        -279
```

Clean up.

```
unloadlibrary shrlibsample
```

See Also

`loadlibrary` | `libfunctionsview`

Concepts

- Passing Arguments
- “MATLAB Crashes Making a Function Call to a Shared Library”

Purpose

Send SOAP message to endpoint

Syntax

```
response = callSoapService(endpoint,soapAction,message)
```

Description

`response = callSoapService(endpoint,soapAction,message)` sends `message`, a Java® document object model (DOM), to the `soapAction` service at `endpoint`. Create `message` using `createSoapMessage`, and extract results from `response` using `parseSoapResponse`.

Examples

This example uses `callSoapService` in conjunction with other SOAP functions to retrieve information about books from a library database, specifically, the author's name for a given book title.

Note The example does not use an actual endpoint; therefore, you cannot run it. The example only illustrates how to use the SOAP functions.

```
% Create the message:
message = createSoapMessage(...
'urn:LibraryCatalog',...
'getAuthor',...
{'In the Fall'},...
{'nameToLookUp'},...
{'{http://www.w3.org/2001/XMLSchema}string'},...
'rpc');
%
% Send the message to the service and get the response:
response = callSoapService(...
'http://test/soap/services/LibraryCatalog',...
'urn:LibraryCatalog#getAuthor',...
message)
%
% Extract MATLAB data from the response
```

```
author = parseSoapResponse(response)
```

MATLAB returns:

```
author = Kate Alvin
```

where author is a char class (type).

See Also

`createClassFromWsdl` | `createSoapMessage` | `parseSoapResponse` | `urlread` | `xmlread`

How To

- "Access Web Services Using MATLAB SOAP Functions"

Purpose

Move camera position and target

Syntax

```
camdolly(dx,dy,dz)
camdolly(dx,dy,dz,'targetmode')
camdolly(dx,dy,dz,'targetmode','coordsys')
camdolly(axes_handle,...)
```

Description

`camdolly(dx,dy,dz)` moves the camera position and the camera target by the specified amounts `dx`, `dy`, and `dz`.

`camdolly(dx,dy,dz,'targetmode')` uses the `targetmode` argument to determine how the camera moves:

- `movetarget` (default) — Move both the camera and the target.
- `fixtarget` — Move only the camera.

`camdolly(dx,dy,dz,'targetmode','coordsys')` uses the `coordsys` argument to determine how MATLAB interprets `dx`, `dy`, and `dz`:

- `camera` (default) — Move in the coordinate system of the camera. `dx` moves left/right, `dy` moves down/up, and `dz` moves along the viewing axis. MATLAB normalizes the units to the scene.

  For example, setting `dx` to 1 moves the camera to the right, which pushes the scene to the left edge of the box formed by the axes position rectangle. A negative value moves the scene in the other direction. Setting `dz` to 0.5 moves the camera to a position halfway between the camera position and the camera target.

- `pixels` — Interpret `dx` and `dy` as pixel offsets and ignore `dz`.
- `data` — Interpret `dx`, `dy`, and `dz` as offsets in axes data coordinates.

`camdolly(axes_handle,...)` operates on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camdolly` operates on the current axes.

`camdolly` sets the axes `CameraPosition` and `CameraTarget` properties, which in turn sets the `CameraPositionMode` and `CameraTargetMode` properties to `manual`.

Examples

Move the camera along the *x*- and *y*-axes in a series of steps:

```
surf(peaks)
axis vis3d
t = 0:pi/20:2*pi;
dx = sin(t)./40;
dy = cos(t)./40;
for i = 1:length(t);
    camdolly(dx(i),dy(i),0)
    drawnow
end
```

See Also

`axes` | `campos` | `camproj` | `camtarget` | `camup` | `camva` | Axes `CameraPosition` property | Axes `CameraTarget` property | Axes `CameraUpVector` property | Axes `CameraViewAngle` property | Axes `Projection` property

How To

- "Defining Scenes with Cameras"

Purpose

Control camera toolbar programmatically

Syntax

```
cameratoolbar
cameratoolbar('NoReset')
cameratoolbar('SetMode',mode)
cameratoolbar('SetCoordSys',coordsys)
cameratoolbar('Show')
cameratoolbar('Hide')
cameratoolbar('Toggle')
cameratoolbar('ResetCameraAndSceneLight')
cameratoolbar('ResetCamera')
cameratoolbar('ResetSceneLight')
cameratoolbar('ResetTarget')
mode = cameratoolbar('GetMode')
paxis = cameratoolbar('GetCoordsys')
vis = cameratoolbar('GetVisible')
cameratoolbar(fig,...)
h = cameratoolbar
cameratoolbar('Close')
```

Description

`cameratoolbar` creates a toolbar that enables interactive manipulation of the axes camera and light when you drag the mouse on the figure window. Several axes camera properties are set when the toolbar is initialized.

`cameratoolbar('NoReset')` creates the toolbar without setting any camera properties.

`cameratoolbar('SetMode',mode)` sets the toolbar mode (depressed button). `mode` can be `'orbit'`, `'orbitscenelight'`, `'pan'`, `'dollyhv'`, `'dollyfb'`, `'zoom'`, `'roll'`, `'nomode'`. For descriptions of the various modes, see “Camera Toolbar”. You can also set these modes using the toolbar, by clicking the respective buttons.

`cameratoolbar('SetCoordSys',coordsys)` sets the principal axis of the camera motion. `coordsys` can be: `'x'`, `'y'`, `'z'`, `'none'`.

`cameratoolbar('Show')` shows the toolbar on the current figure.

`cameratoolbar('Hide')` hides the toolbar on the current figure.

`cameratoolbar('Toggle')` toggles the visibility of the toolbar.

`cameratoolbar('ResetCameraAndSceneLight')` resets the current camera and scenelight.

`cameratoolbar('ResetCamera')` resets the current camera.

`cameratoolbar('ResetSceneLight')` resets the current scenelight.

`cameratoolbar('ResetTarget')` resets the current camera target.

`mode = cameratoolbar('GetMode')` returns the current mode.

`paxis = cameratoolbar('GetCoordsys')` returns the current principal axis.

`vis = cameratoolbar('GetVisible')` returns the visibility of the toolbar (`1` if visible, `0` if not visible).

`cameratoolbar(fig,...)` specifies the figure to operate on by passing the figure handle as the first argument.

`h = cameratoolbar` returns the handle to the toolbar.

`cameratoolbar('Close')` removes the toolbar from the current figure.

In general, the use of OpenGL hardware improves rendering performance.

Alternatives

Display the toolbar by selecting **Camera Toolbar** from the figure window's **View** menu.

See Also

`rotate3d` | `zoom`

How To

- "Camera Toolbar"

Purpose

Create or move light object in camera coordinates

Syntax

```
camlight('headlight')
camlight('right')
camlight('left')
camlight
camlight(az,el)
camlight(...,'style')
camlight(light_handle,...)
light_handle = camlight(...)
```

Description

`camlight('headlight')` creates a light at the camera position.

`camlight('right')` creates a light right and up from camera.

`camlight('left')` creates a light left and up from camera.

`camlight` with no arguments is the same as `camlight('right')`.

`camlight(az,el)` creates a light at the specified azimuth (`az`) and elevation (`el`) with respect to the camera position. The camera target is the center of rotation and `az` and `el` are in degrees.

`camlight(...,'style')` defines the style argument using one of two values:

- `local` (default) — The light is a point source that radiates from the location in all directions.
- `infinite` — The light shines in parallel rays.

`camlight(light_handle,...)` uses the light specified in `light_handle`.

`light_handle = camlight(...)` returns the light object handle.

`camlight` sets the light object `Position` and `Style` properties. A light created with `camlight` does not track the camera. In order for the light to stay in a constant position relative to the camera, call `camlight` whenever you move the camera.

Examples

Create a light positioned to the left of the camera and then reposition the light each time the camera moves:

```
surf(peaks)
axis vis3d
h = camlight('left');
for i = 1:20;
 camorbit(10,0)
 camlight(h,'left')
 drawnow;
end
```

See Also

`lightangle` | `light`

How To

- “Lighting Overview”

Purpose

Position camera to view object or group of objects

Syntax

```
camlookat(object_handles)
camlookat(axes_handle)
camlookat
```

Description

`camlookat(object_handles)` views the objects identified in the vector `object_handles`. The vector can contain the handles of axes children.

`camlookat(axes_handle)` views the objects that are children of the axes identified by `axes_handle`.

`camlookat` views the objects that are in the current axes by moving the camera position and camera target while preserving the relative view direction and camera view angle. The viewed object (or objects) roughly fill the axes position rectangle. To change the view, `camlookat` sets the axes `CameraPosition` and `CameraTarget` properties.

Examples

Create three spheres at different locations and then progressively position the camera so that the scene composes around each sphere individually:

```
% Create three spheres using the sphere function:
[x y z] = sphere;
s1 = surf(x,y,z);
hold on
s2 = surf(x+3,y,z+3);
s3 = surf(x,y,z+6);
% Set the data aspect ratio using daspect:
daspect([1 1 1])
% Set the view:
view(30,10)
% Set the projection type using camproj:
camproj perspective
% Compose the scene around the current axes
camlookat(gca)
pause(2)
% Compose the scene around sphere s1
```

```
camlookat(s1)
pause(2)
% Compose the scene around sphere s2
camlookat(s2)
pause(2)
% Compose the scene around sphere s3
camlookat(s3)
pause(2)
camlookat(gca)
```

See Also

campos | camtarget

How To

- "Defining Scenes with Cameras"

Purpose

Rotate camera position around camera target

Syntax

```
camorbit(dtheta,dphi)
camorbit(dtheta,dphi,'coordsys')
camorbit(dtheta,dphi,'coordsys','direction')
camorbit(axes_handle,...)
```

Description

`camorbit(dtheta,dphi)` rotates the camera position around the camera target by the amounts specified in `dtheta` and `dphi` (both in degrees). `dtheta` is the horizontal rotation and `dphi` is the vertical rotation.

`camorbit(dtheta,dphi,'`*`coordsys`*`')` rotates the camera position around the camera target, using the *`coordsys`* argument to determine the center of rotation. *`coordsys`* can take on two values:

- `data` (default) — Rotate the camera around an axis defined by the camera target and the `direction` (default is the positive *z* direction).
- `camera` — Rotate the camera about the point defined by the camera target.

`camorbit(dtheta,dphi,'`*`coordsys`*`','direction')` defines the axis of rotation for the data coordinate system using the `direction` argument in conjunction with the camera target. Specify `direction` as a three-element vector containing the *x*-, *y*-, and *z*-components of the direction or one of the characters, `x`, `y`, or `z`, to indicate `[1 0 0]`, `[0 1 0]`, or `[0 0 1]` respectively.

`camorbit(axes_handle,...)` operates on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camorbit` operates on the current axes.

The behavior of `camorbit` differs from the `rotate3d` function in that while the `rotate3d` tool modifies the `View` property of the axes, the `camorbit` function fixes the aspect ratio and modifies the `CameraTarget`, `CameraPosition` and `CameraUpVector` properties of the axes. See `Axes Properties` for more information.

Examples

Rotate the camera horizontally about a line defined by the camera target point and a direction that is parallel to the *y*-axis. Visualize this rotation as a cone formed with the camera target at the apex and the camera position forming the base:

```
surf(peaks)
axis vis3d
for i=1:36
   camorbit(10,0,'data',[0 1 0])
   drawnow
end
```

Rotate in the `camera` coordinate system to orbit the camera around the axes along a circle while keeping the center of a circle at the camera target:

```
surf(peaks)
axis vis3d
for i=1:36
   camorbit(10,0,'camera')
   drawnow
end
```

Alternatives

Enable 3-D rotation from the figure **Tools** menu or the figure toolbar.

See Also

`axes` | `axis` | `camdolly` | `campan` | `camzoom` | `camroll`

How To

- “Defining Scenes with Cameras”

Purpose

Rotate camera target around camera position

Syntax

```
campan(dtheta,dphi)
campan(dtheta,dphi,coordsys)
campan(dtheta,dphi,coordsys,direction)
campan(axes_handle,...)
```

Description

`campan(dtheta,dphi)` rotates the camera target of the current axes around the camera position by the amounts specified in `dtheta` and `dphi` (both in degrees). `dtheta` is the horizontal rotation and `dphi` is the vertical rotation.

`campan(dtheta,dphi,coordsys)` determine the center of rotation using the `coordsys` argument. It can take on two values:

- `'data'` (default) — Rotate the camera target around an axis defined by the camera position and the `direction` (default is the positive *z* direction)
- `'camera'` — Rotate the camera about the point defined by the camera target.

`campan(dtheta,dphi,coordsys,direction)` defines the axis of rotation for the data coordinate system using the `direction` argument with the camera position. Specify `direction` as a three-element vector containing the *x*-, *y*-, and *z*-components of the direction or one of the characters, `'x'`, `'y'`, or `'z'`, to indicate `[1 0 0]`, `[0 1 0]`, or `[0 0 1]` respectively.

`campan(axes_handle,...)` operates on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `campan` operates on the current axes.

Examples

Move the camera target to pan the object in a circular motion.

```
sphere;
axis vis3d
hPan = sin(-pi:1:pi);
vPan = cos(-pi:1:pi);
```

```
for k=1:length(hPan)
   campan(hPan(k),vPan(k))
   drawnow
   pause(.1)
end
```

See Also

axes | camdolly | camorbit | camtarget | camzoom | camroll

How To

- "Defining Scenes with Cameras"

Purpose

Set or query camera position

Syntax

```
campos
campos([camera_position])
campos('mode')
campos('auto')
campos('manual')
campos(axes_handle,...)
```

Description

`campos` returns the camera position in the current axes.

`campos([camera_position])` sets the position of the camera in the current axes to the specified value. Specify the position as a three-element vector containing the *x*-, *y*-, and *z*-coordinates of the desired location in the data units of the axes.

`campos('mode')` returns the value of the camera position mode, which can be either `auto` (the default) or `manual`.

`campos('auto')` sets the camera position mode to `auto`.

`campos('manual')` sets the camera position mode to `manual`.

`campos(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `campos` operates on the current axes.

`campos` sets or queries values of the axes `CameraPosition` and `CameraPositionMode` properties. The camera position is the point in the Cartesian coordinate system of the axes from which you view the scene.

Examples

Move the camera along the *x*-axis in a series of steps:

```
surf(peaks)
axis vis3d off
for x = -200:5:200
    campos([x,5,10])
    drawnow
end
```

See Also

`axis` | `camproj` | `camtarget` | `camup` | `camva` | Axes: `CameraPosition` | Axes: `CameraTarget` | Axes: `CameraUpVector` | Axes: `CameraViewAngle` | Axes: `Projection`

How To

- "Defining Scenes with Cameras"

Purpose

Set or query projection type

Syntax

```
camproj
camproj('projection_type')
camproj(axes_handle,...)
```

Description

`camproj` returns the projection type setting in the current axes. The projection type determines whether MATLAB 3-D views use a perspective or orthographic projection.

`camproj('projection_type')` sets the projection type in the current axes to the specified value. Possible values for *`projection_type`* are `orthographic` and `perspective`.

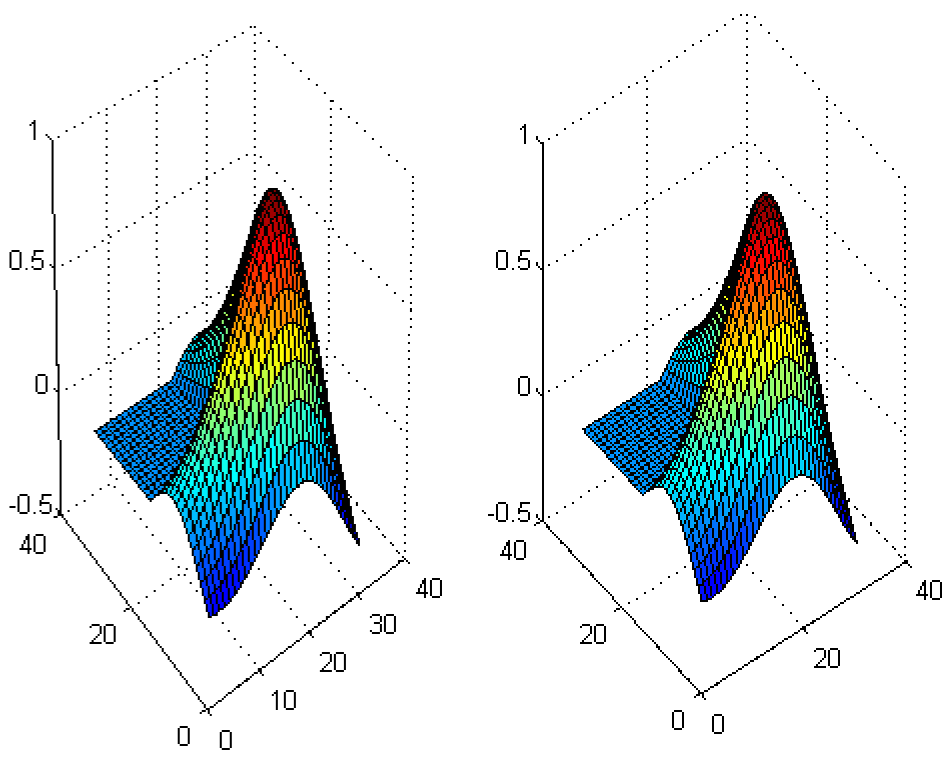
`camproj(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camproj` operates on the current axes.

`camproj` sets or queries values of the axes object `Projection` property.

Examples

Compare the different `camproj` settings using `subplot`:

```
subplot(1,2,1); surf(membrane); camproj('perspective')
subplot(1,2,2); surf(membrane); camproj('orthographic')
```



See Also

`axis` | `campos` | `camtarget` | `camup` | `camva` | Axes: `CameraPosition` | Axes: `CameraTarget` | Axes: `CameraUpVector` | Axes: `CameraViewAngle` | Axes: `Projection`

How To

- "Defining Scenes with Cameras"

Purpose

Rotate camera about view axis

Syntax

```
camroll(dtheta)
camroll(axes_handle,dtheta)
```

Description

`camroll(dtheta)` rotates the camera around the camera viewing axis by the amounts specified in `dtheta` (in degrees). The viewing axis is the line passing through the camera position and the camera target.

`camroll(axes_handle,dtheta)` operates on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camroll` operates on the current axes.

`camroll` sets the axes `CameraUpVector` property and also sets the `CameraUpVectorMode` property to `manual`.

Examples

Rotate the camera around the viewing axis:

```
surf(peaks)
axis vis3d
for i=1:36
 camroll(10)
 drawnow
end
```

See Also

`axes` | `axis` | `camdolly` | `camorbit` | `camzoom` | `campan`

How To

- “Defining Scenes with Cameras”

camtarget

Purpose

Set or query location of camera target

Syntax

```
camtarget
camtarget([camera_target])
camtarget('mode')
camtarget('auto')
camtarget('manual')
camtarget(axes_handle,...)
```

Description

`camtarget` returns the location of the camera target in the current axes. The camera target is the location in the axes that the camera points to. The camera remains oriented toward this point regardless of its position.

`camtarget([camera_target])` sets the camera target in the current axes to the specified value. Specify the target as a three-element vector containing the *x*-, *y*-, and *z*-coordinates of the desired location in the data units of the axes.

`camtarget('mode')` returns the value of the camera target mode, which can be either `auto` (default) or `manual`.

`camtarget('auto')` sets the camera target mode to `auto`. When the camera target mode is `auto`, the camera target is the center of the axes plot box.

`camtarget('manual')` sets the camera target mode to `manual`.

`camtarget(axes_handle,...)` performs the set or query on the axes identified by `axes_handle`. When you do not specify an axes handle, `camtarget` operates on the current axes.

`camtarget` sets or queries values of the axes object `CameraTarget` and `CameraTargetMode` properties.

Examples

Move the camera position and the camera target along the *x*-axis in a series of steps:

```
surf(peaks);
axis vis3d
```

```
xp = linspace(-150,40,50);
xt = linspace(25,50,50);
for i=1:50
     campos([xp(i),25,5]);
     camtarget([xt(i),30,0])
     drawnow
end
```

See Also

axis | campos | camup | camva | Axes: CameraPosition | Axes: CameraTarget | Axes: CameraUpVector | Axes: CameraViewAngle | Axes: Projection

How To

- "Defining Scenes with Cameras"

camup

Purpose

Set or query camera up vector

Syntax

```
camup
camup([up_vector])
camup('mode')
camup('auto')
camup('manual')
camup(axes_handle,...)
```

Description

`camup` returns the camera up vector setting in the current axes. The camera up vector specifies the direction that is oriented up in the scene.

`camup([up_vector])` sets the up vector in the current axes to the specified value. Specify the up vector as *x*, *y*, and *z* components.

`camup('mode')` returns the current value of the camera up vector mode, which can be either `auto` (default) or `manual`.

`camup('auto')` sets the camera up vector mode to `auto`. In `auto` mode, `[0 1 0]` is the up vector of for 2-D views. This means the *y*-axis points up. For 3-D views, the up vector is `[0 0 1]`, meaning the *z*-axis points up.

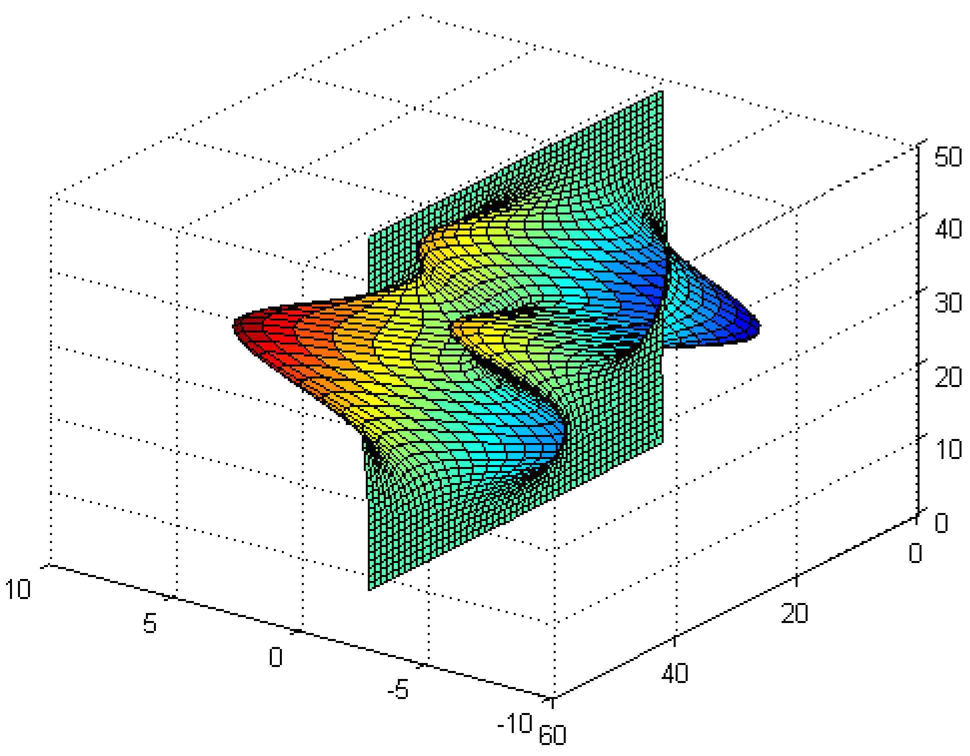
`camup('manual')` sets the camera up vector mode to `manual`. In `manual` mode, the value of the camera up vector does not change unless you set it.

`camup(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camup` operates on the current axes.

Examples

Set the *x*-axis to be the up axis:

```
surf(peaks)
camup([1 0 0]);
```



See Also

axis | campos | camup | camtarget | Axes: CameraPosition | Axes: CameraTarget | Axes: CameraUpVector | Axes: CameraViewAngle | Axes: Projection

How To

- “Defining Scenes with Cameras”

Purpose

Set or query camera view angle

Syntax

```
camva
camva(view_angle)
camva('mode')
camva('auto')
camva('manual')
camva(axes_handle,...)
```

Description

`camva` returns the camera view angle setting in the current axes. The camera view angle determines the field of view of the camera. Larger angles produce a smaller view of the scene. Implement zooming by changing the camera view angle.

`camva(view_angle)` sets the view angle in the current axes to the specified value. Specify the view angle in degrees.

`camva('mode')` returns the current value of the camera view angle mode, which can be either `auto` (the default) or `manual`.

`camva('auto')` sets the camera view angle mode to `auto`.

`camva('manual')` sets the camera view angle mode to `manual`.

`camva(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camva` operates on the current axes.

Tips

The `camva` function sets or queries values of the axes object `CameraViewAngle` and `CameraViewAngleMode` properties.

When the camera view angle mode is `auto`, the camera view angle adjusts so that the scene fills the available space in the window. If you move the camera to a different position, the camera view angle changes to maintain a view of the scene that fills the available area in the window.

Setting a camera view angle or setting the camera view angle to `manual` disables the MATLAB stretch-to-fill feature (stretching of the axes

to fit the window). This means setting the camera view angle to its current value,

```
camva(camva)
```

can cause a change in the way the graph looks. See `axes` for more information.

Examples

Create two pushbuttons, one that zooms in and another that zooms out:

```
% Set the range checking in the callback statements to keep
% the values for the camera view angle in the range greater
% than zero and less than 180.
uicontrol('Style','pushbutton',...
  'String','Zoom In',...
  'Position',[20 20 60 20],...
  'Callback','if camva <= 1;return;else;camva(camva-1);end');
uicontrol('Style','pushbutton',...
  'String','Zoom Out',...
  'Position',[100 20 60 20],...
  'Callback','if camva >= 179;return;else;camva(camva+1);end');
% Now create a graph to zoom in and out on:
surf(peaks);
```

See Also

`axis` | `campos` | `camup` | `camtarget` | Axes: `CameraPosition` | Axes: `CameraTarget` | Axes: `CameraUpVector` | Axes: `CameraViewAngle` | Axes: `Projection`

How To

- "Defining Scenes with Cameras"

Purpose

Zoom in and out on scene

Syntax

```
camzoom(zoom_factor)
camzoom(axes_handle,...)
```

Description

`camzoom(zoom_factor)` zooms in or out on the scene depending on the value specified by `zoom_factor`. If `zoom_factor` is greater than 1, the scene appears larger; if `zoom_factor` is greater than zero and less than 1, the scene appears smaller.

`camzoom(axes_handle,...)` operates on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `camzoom` operates on the current axes.

Tips

`camzoom` sets the axes `CameraViewAngle` property, which in turn causes the `CameraViewAngleMode` property to be set to `manual`. Note that setting the `CameraViewAngle` property disables the MATLAB stretch-to-fill feature (stretching of the axes to fit the window). This may result in a change to the aspect ratio of your graph. See the `axes` function for more information on this behavior.

See Also

`axes` | `camdolly` | `camorbit` | `campan` | `camroll` | `camva`

How To

- "Defining Scenes with Cameras"

Purpose

(Will be removed) Convert point coordinates from cartesian to barycentric

Note `cartToBary(TriRep)` will be removed in a future release. Use `cartesianToBarycentric(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
B = cartToBary(TR, SI, XC)
```

Description

`B = cartToBary(TR, SI, XC)` returns the barycentric coordinates of each point in `XC` with respect to its associated simplex `SI`.

Input Arguments

`TR`	Triangulation representation.
`SI`	Column vector of simplex indices that index into the triangulation matrix `TR.Triangulation`.
`XC`	Matrix that represents the Cartesian coordinates of the points to be converted. `XC` is of size `m`-by-`n`, where `m` is of `length(SI)`, the number of points to convert, and `n` is the dimension of the space where the triangulation resides.

Output Arguments

`B`	Matrix of dimension `m`-by-`k` where `k` is the number of vertices per simplex.

Definitions

A simplex is a triangle/tetrahedron or higher dimensional equivalent.

Examples

Compute the Delaunay triangulation of a set of points.

```
x = [0 4 8 12 0 4 8 12]';
```

```
y = [0 0 0 0 8 8 8 8]';
dt = DelaunayTri(x,y)
```

Compute the barycentric coordinates of the incenters.

```
cc = incenters(dt);
tri = dt(:,:);
```

Plot the original triangulation and reference points.

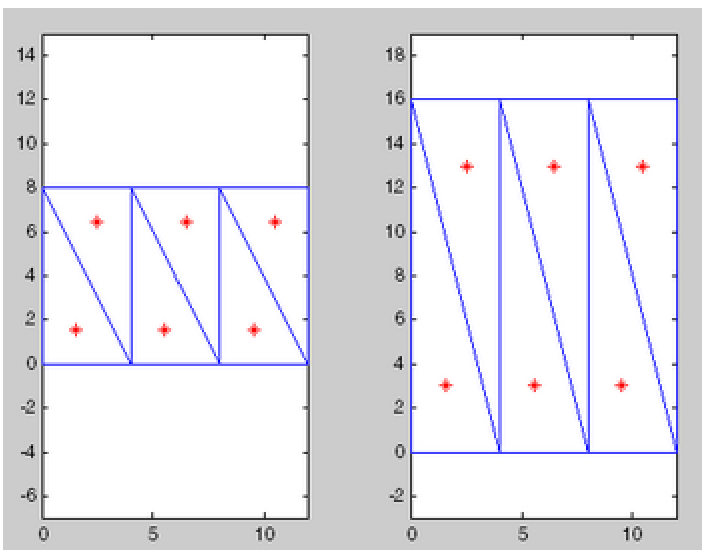
```
figure
subplot(1,2,1);
triplot(dt); hold on;
plot(cc(:,1), cc(:,2), '*r');
hold off;
axis equal;
```

Stretch the triangulation and compute the mapped locations of the incenters on the deformed triangulation.

```
b = cartToBary(dt,[1:length(tri)]',cc);
y = [0 0 0 0 16 16 16 16]';
tr = TriRep(tri,x,y)
xc = baryToCart(tr, [1:length(tri)]', b);
```

Plot the deformed triangulation and mapped locations of the reference points.

```
subplot(1,2,2);
triplot(tr);
hold on;
plot(xc(:,1), xc(:,2), '*r');
hold off;
axis equal;
```



See Also

barycentricToCartesian | pointLocation | delaunayTriangulation | triangulation

cart2pol

Purpose

Transform Cartesian coordinates to polar or cylindrical

Syntax

```
[THETA,RHO,Z] = cart2pol(X,Y,Z)
[THETA,RHO] = cart2pol(X,Y)
```

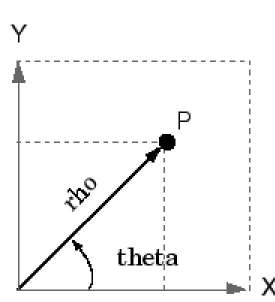
Description

`[THETA,RHO,Z] = cart2pol(X,Y,Z)` transforms three-dimensional Cartesian coordinates stored in corresponding elements of arrays `X`, `Y`, and `Z`, into cylindrical coordinates. `THETA` is a counterclockwise angular displacement in radians from the positive *x*-axis, `RHO` is the distance from the origin to a point in the *x*-*y* plane, and `Z` is the height above the *x*-*y* plane. Arrays `X`, `Y`, and `Z` must be the same size (or any can be scalar).

`[THETA,RHO] = cart2pol(X,Y)` transforms two-dimensional Cartesian coordinates stored in corresponding elements of arrays `X` and `Y` into polar coordinates.

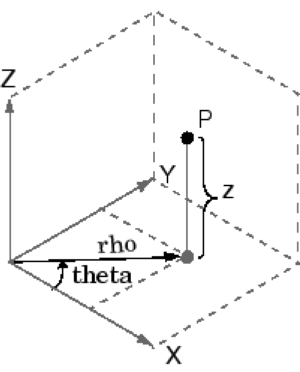
Algorithms

The mapping from two-dimensional Cartesian coordinates to polar coordinates, and from three-dimensional Cartesian coordinates to cylindrical coordinates is



Two-Dimensional Mapping

```
theta = atan2(y,x)
rho = sqrt(x.^2 + y.^2)
```



Three-Dimensional Mapping

```
theta = atan2(y,x)
rho = sqrt(x.^2 + y.^2)
z = z
```

See Also

`cart2sph` | `pol2cart` | `sph2cart`

cart2sph

Purpose

Transform Cartesian coordinates to spherical

Syntax

```
[azimuth,elevation,r] = cart2sph(X,Y,Z)
```

Description

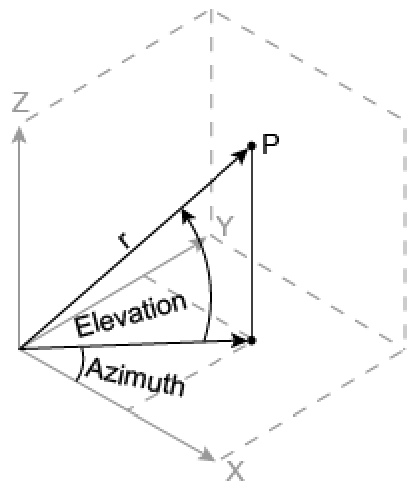
`[azimuth,elevation,r] = cart2sph(X,Y,Z)` transforms Cartesian coordinates stored in corresponding elements of arrays `X`, `Y`, and `Z` into spherical coordinates. `azimuth` and `elevation` are angular displacements in radians measured from the positive *x*-axis, and the *x*-*y* plane, respectively; and `r` is the distance from the origin to a point.

Arrays `X`, `Y`, and `Z` must be the same size (or any of them can be scalar).

Algorithms

The mapping from three-dimensional Cartesian coordinates to spherical coordinates is

```
azimuth = atan2(y,x)
elevation = atan2(z,sqrt(x.^2 + y.^2))
r = sqrt(x.^2 + y.^2 + z.^2)
```



The notation for spherical coordinates is not standard. For the `cart2sph` function, `elevation` is measured from the *x*-*y* plane. Notice that if

`elevation = 0`, the point is in the *x*-*y* plane. If `elevation = pi/2`, then the point is on the positive *z*-axis.

See Also

`cart2pol` | `pol2cart` | `sph2cart`

case

Purpose

Optional keyword in switch statement

Syntax

```
switch switch_expression
   case case_expression
      statements
   case case_expression
      statements
    :
   otherwise
      statements
end
```

Description

A `switch` block conditionally executes one set of statements from several choices. Each choice is a `case`.

An evaluated *`switch_expression`* is a scalar or string. An evaluated *`case_expression`* is a scalar, a string, or a cell array of scalars or strings. The `switch` block tests each case until one of the cases is `true`. A case is `true` when:

- For numbers, `eq(`*`case_expression`*`,`*`switch_expression`*`)`.
- For strings, `strcmp(`*`case_expression`*`,`*`switch_expression`*`)`.
- For objects that support the `eq` function, `eq(`*`case_expression`*`,`*`switch_expression`*`)`.
- For a cell array *`case_expression`*, at least one of the elements of the cell array matches *`switch_expression`*, as defined above for numbers, strings, and objects.

When a case is true, MATLAB executes the corresponding statements, and then exits the `switch` block.

`otherwise` is optional, and executes only when no case is true.

For more information, see the `switch` reference page.

Examples

Decide which plot to create based on the value of the string `plottype`:

```
x = [12, 64, 24];
plottype = 'pie3';

switch plottype
    case 'bar'
        bar(x)
        title('Bar Graph')
    case {'pie','pie3'}
        pie3(x)
        title('Pie Chart')
        legend('First','Second','Third')
    otherwise
        warning('Unexpected plot type. No plot created.');
end
```

See Also

`switch` | `end` | `if` | `for` | `while`

Purpose

Cast variable to different data type

Syntax

```
B = cast(A,newclass)
B = cast(A,'like',p)
```

Description

`B = cast(A,newclass)` converts `A` to class `newclass`, where `newclass` is the name of a built-in data type compatible with `A`. The `cast` function truncates any values in `A` that are too large to map into `newclass`.

`B = cast(A,'like',p)` converts `A` to the same data type and sparsity as the variable `p`. If `A` and `p` are both real, then `B` is also real. Otherwise, `B` is complex.

Examples

Convert Numeric Data Type

Convert an `int8` value to `uint8`.

Define a scalar 8-bit integer.

```
a = int8(5);
```

Convert `a` to an unsigned 8-bit integer.

```
b = cast(a,'uint8');
class(b)
```

```
ans =
uint8
```

Match Data Type and Complex Nature of `p`

Define a single precision vector `p` that is complex valued.

```
p = single([1+i 2]);
```

Define a 2-by-3 matrix of ones.

```
A = ones(2,3);
```

Convert `A` to the same data type and complexity (real or complex) as `p`.

```
B = cast(A,'like',p)
```

```
B =

   1.0000 + 0.0000i   1.0000 + 0.0000i   1.0000 + 0.0000i
   1.0000 + 0.0000i   1.0000 + 0.0000i   1.0000 + 0.0000i
```

```
class(B)
```

```
ans =

single
```

See Also

`class` | `typecast`

Purpose

Concatenate arrays along specified dimension

Syntax

```
C = cat(dim, A, B)
C = cat(dim, A1, A2, A3, A4, ...)
```

Description

`C = cat(dim, A, B)`concatenates the arrays `A` and `B` along array dimension `dim`.

`C = cat(dim, A1, A2, A3, A4, ...)` concatenates all the input arrays (`A1`, `A2`, `A3`, `A4`, and so on) along array dimension `dim`.

For nonempty arrays, `cat(2, A, B)` is the same as `[A, B]`, and `cat(1, A, B)` is the same as `[A; B]`.

Tips

When used with comma-separated list syntax, `cat(dim, C{:})` or `cat(dim, C.field)` is a convenient way to concatenate a cell or structure array containing numeric matrices into a single matrix.

For information on combining unlike integer types, integers with nonintegers, cell arrays with non-cell arrays, or empty matrices with other elements, see "Valid Combinations of Unlike Classes"

Examples

Given

```
A =                     B =
      1     2                          5     6
      3     4                          7     8
```

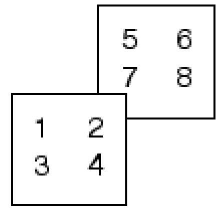
concatenating along different dimensions produces

1 2
3 4
5 6
7 8

`C = cat(1,A,B)`

1 2 5 6
3 4 7 8

`C = cat(2,A,B)`



`C = cat(3,A,B)`

The commands

```
A = magic(3); B = pascal(3);
C = cat(4, A, B);
```

produce a 3-by-3-by-1-by-2 array.

See Also

`vertcat` | `horzcat` | `strcat` | `char` | `num2cell` | `reshape` | `squeeze` | `strjoin` | `shiftdim` | `special character`

catch

Purpose

Handle error detected in try statement

Syntax

```
try
   statements
catch exception
   statements
end
```

Description

`try` and `catch` blocks allow you to override the default error behavior for a set of program statements. If any statement in a `try` block generates an error, program control goes immediately to the `catch` block, which contains your error handling statements.

`exception` is an optional `MException` object input to the `catch` block that allows you to identify the error.

Both `try` and `catch` blocks can contain nested `try/catch` statements.

Examples

Provide more information about a dimension mismatch error:

```
A = rand(3);
B = ones(5);

try
   C = [A; B];
catch err

   % Give more information for mismatch.
   if (strcmp(err.identifier,'MATLAB:catenate:dimensionMismatch'))

      msg = sprintf('%s', ...
            'Dimension mismatch occured: First argument has ', ...
             num2str(size(A,2)), ' columns while second has ', ...
             num2str(size(B,2)), ' columns.');
      error('MATLAB:myCode:dimensions', msg);

   % Display any other errors as usual.
   else
```

```
      rethrow(err);
   end

end  % end try/catch
```

See Also

error | assert | try | MException

caxis

Purpose

Color axis scaling

Syntax

```
caxis([cmin cmax])
caxis auto
caxis manual
caxis(caxis) freeze
v = caxis
caxis(axes_handle,...)
```

Description

`caxis` controls the mapping of data values to the colormap. It affects any surfaces, patches, and images with indexed `CData` and `CDataMapping` set to scaled. It does not affect surfaces, patches, or images with true color `CData` or with `CDataMapping` set to `direct`.

`caxis([cmin cmax])` sets the color limits to specified minimum and maximum values. Data values less than `cmin` or greater than `cmax` map to `cmin` and `cmax`, respectively. Values between `cmin` and `cmax` linearly map to the current colormap.

`caxis auto` computes the color limits automatically using the minimum and maximum data values. This is the default behavior. Color values set to `Inf` map to the maximum color, and values set to `-Inf` map to the minimum color. Faces or edges with color values set to `NaN` are not drawn.

`caxis manual` and `caxis(caxis) freeze` the color axis scaling at the current limits. This enables subsequent plots to use the same limits when `hold` is `on`.

`v = caxis` returns a two-element row vector containing the `[cmin cmax]` currently in use.

`caxis(axes_handle,...)` uses the axes specified by `axes_handle` instead of the current axes.

`caxis` changes the `CLim` and `CLimMode` properties of axes graphics objects.

Tips

How Color Axis Scaling Works

Surface, patch, and image graphics objects having indexed `CData` and `CDataMapping` set to `scaled` map `CData` values to colors in the figure colormap each time they render. `CData` values equal to or less than `cmin` map to the first color value in the colormap, and `CData` values equal to or greater than `cmax` map to the last color value in the colormap. The following linear transformation is performed on the intermediate values (referred to as `C` below) to map them to an entry in the colormap (whose length is `m`, and whose row index is referred to as `index` below).

```
index = fix((C-cmin)/(cmax-cmin)*m)+1;
%Clamp values outside the range [1 m]
index(index<1) = 1;
index(index>m) = m;
```

Examples

Create `(X,Y,Z)` data for a sphere and view the data as a surface.

```
[X,Y,Z] = sphere;
C = Z;
surf(X,Y,Z,C)
```

Values of `C` have the range [-1 1]. Values of `C` near -1 are assigned the lowest values in the colormap; values of `C` near 1 are assigned the highest values in the colormap.

To map the top half of the surface to the highest value in the color table, use

```
caxis([-1 0])
```

To use only the bottom half of the color table, enter

```
caxis([-1 3])
```

which maps the lowest `CData` values to the bottom of the colormap, and the highest values to the middle of the colormap (by specifying a `cmax` whose value is equal to `cmin` plus twice the range of the `CData`).

The command

```
caxis auto
```

resets axis scaling back to autoranging and you see all the colors in the surface. In this case, entering

```
caxis
```

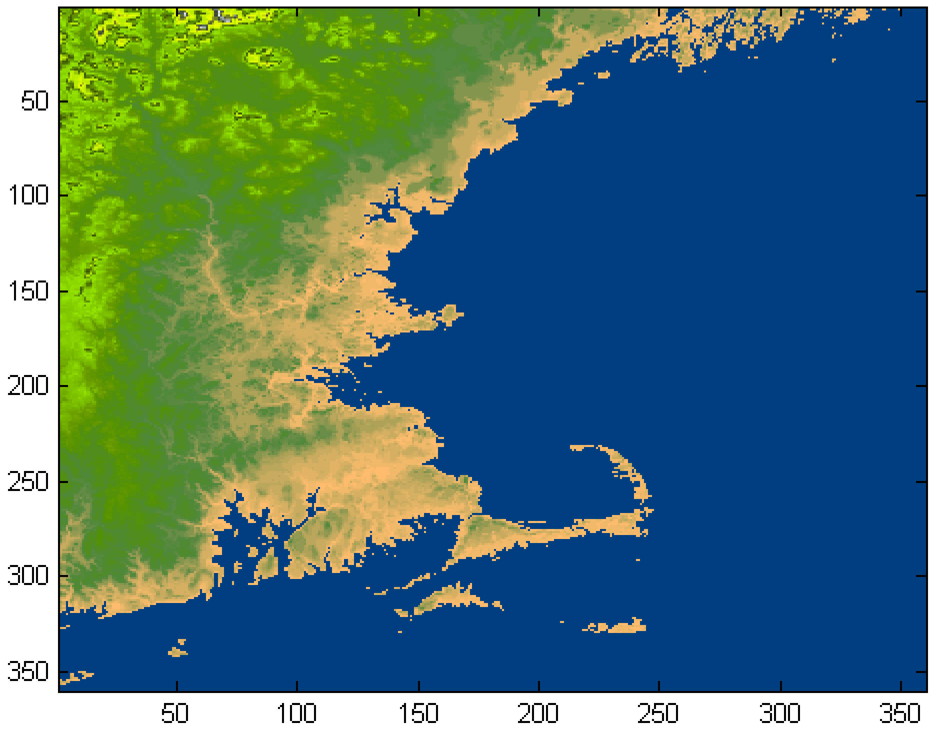
returns

```
[-1 1]
```

Using caxis with Image Data

Adjusting the color axis can be useful when using images with scaled color data. For example, load the image data and colormap for Cape Cod, Massachusetts. This command loads the image's data `X` and the image's colormap `map` into the workspace.

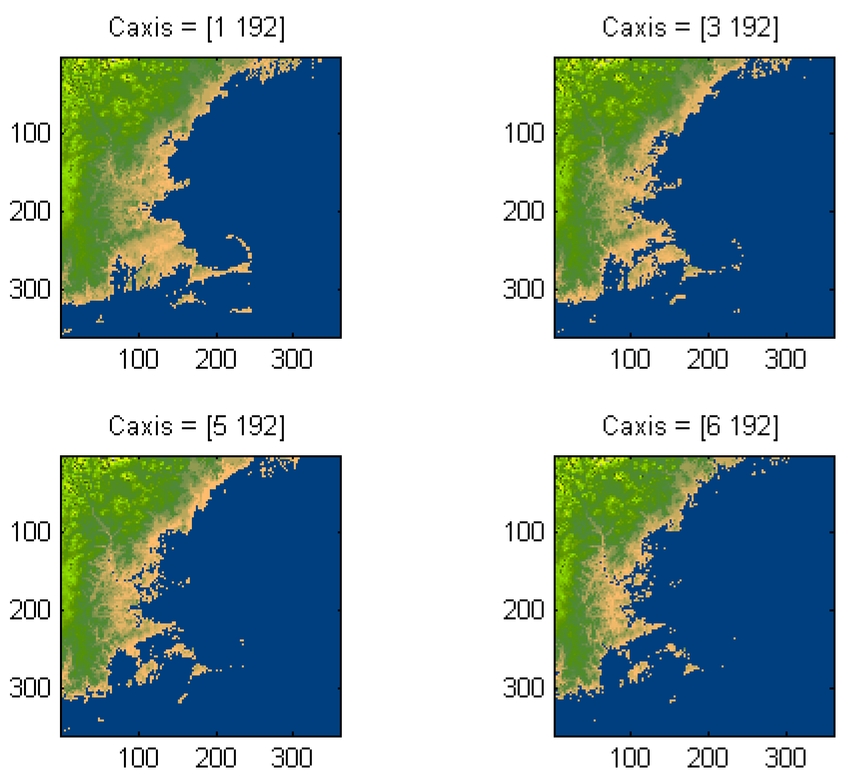
```
load cape
% Display the image with CDataMapping set to scaled and
%  install the image's colormap.
image(X,'CDataMapping','scaled')
colormap(map)
% This adjusts the color limits to span the range
% of the image data, which is 1 to 192:
caxis
ans =
     1   192
```



The blue color of the ocean is the first color in the colormap and is mapped to the lowest data value (1). You can effectively move sea level by changing the lower color limit value. For example, The following code compares four settings:

```
figure
load cape
colormap(map)
subplot(2,2,1)
image(X,'CDataMapping','scaled')
```

```
axis image
title('Caxis = [1 192]')
subplot(2,2,2)
image(X,'CDataMapping','scaled')
axis image
title('Caxis = [3 192]')
caxis([3 192])
subplot(2,2,3)
image(X,'CDataMapping','scaled')
axis image
title('Caxis = [5 192]')
caxis([5 192])
subplot(2,2,4)
image(X,'CDataMapping','scaled')
axis image
title('Caxis = [6 192]')
caxis([6 192])
```



See Also

`axes` | `axis` | `colormap` | `get` | `mesh` | `pcolor` | `set` | `surf` | CLim | Colormap | CLimMode

-
- "Axes Color Limits — the CLim Property"

Purpose

Change current folder

Syntax

```
cd(newFolder)
oldFolder = cd(newFolder)
cd
```

Description

`cd(newFolder)` changes the current folder to `newFolder`.

`oldFolder = cd(newFolder)` returns the existing current folder as a string to `oldFolder`, and then changes the current folder to `newFolder`.

`cd` displays the current folder.

Tips

- When using the command syntax (`cd newfolder`), if the `newFolder` string contains spaces, enclose the string in single quotation marks.
- On UNIX platforms, use the ~ (tilde) character to represent the user home directory.
- If you use `cd` within a local function, the folder change persists after program control returns from the function. That is, the scope of the folder change is global.

Input Arguments

newFolder

A string specifying the folder to which you want to change the current folder. Valid values can be any one of the following:

- A full or relative path.
- `../`, which indicates one level up from the current folder.
- Multiple strings of `../`, which indicates multiple levels up from the current folder.
- `./`, which indicates a path relative to the current folder, although without the `./`, `cd` assumes that the path is relative to the current folder.

Output Arguments

oldFolder

A string specifying the current folder that was in place when you issued the `cd` command.

Definitions

The current folder is a reference location that MATLAB uses to find files. See "The Current Folder".

Examples

Use `cd` with the `matlabroot` function to change the current folder to the examples directory for the currently running version of MATLAB:

```
cd(fullfile(matlabroot, '/help/techdoc/matlab_env/examples'))
```

On a Microsoft Windows platform, specify the full path to change the current folder from any location to the examples directory for MATLAB Version 7.11 (R2010b), assuming that version is installed on your `C:` drive:

```
cd('C:/Program Files/MATLAB/R2010b/help/techdoc/matlab_env/examples')

% Change the current folder from
% C:/Program Files/MATLAB/R2010b/help/techdoc/matlab_env/examples to
% C:/Program Files/MATLAB/R2010b/help/techdoc:

cd ../..

% Use a relative path to change the current folder from
% C:/Program Files/MATLAB/R2010b/help/techdoc back to
% C:/Program Files/MATLAB/R2010b/help/techdoc/matlab_env/examples:

cd matlab_env/examples

% Change the current folder from its current location to a new locatio
% but save its previous location. Later, change the current folder to
% previous location.
```

```
% This returns C:/Program Files/MATLAB/R2010b/help/techdoc/matlab_env/exa
% to oldFolder, and then changes the current folder to C:/Program Files:

oldFolder = cd('C:/Program Files')

% Display current folder:

pwd

% Change the current folder to the previous location:
cd(oldFolder)

pwd
```

On a UNIX platform, change the current folder to the examples directory for the currently running version of MATLAB, assuming it is installed in your home location:

```
cd('~/help/techdoc/matlab_env/examples')
```

See Also

`dir` | `fileparts` | `path` | `pwd` | `what`

How To

- "Path Names in MATLAB"
- "Files and Folders that MATLAB Accesses"

Purpose

(Will be removed) Convex hull

Note `convexHull(DelaunayTri)` will be removed in a future release. Use `convexHull(delaunayTriangulation)` instead.

`DelaunayTri` will be removed in a future release. Use `delaunayTriangulation` instead.

Syntax

```
K = convexHull(DT)
[K AV] = convexHull(DT)
```

Description

`K = convexHull(DT)` returns the indices into the array of points `DT.X` that correspond to the vertices of the convex hull.

`[K AV] = convexHull(DT)` returns the convex hull and the area or volume bounded by the convex hull.

Input Arguments

`DT`	Delaunay triangulation.

Output Arguments

`K`	If the points lie in 2-D space, `K` is a column vector of length `numf`. Otherwise `K` is a matrix of size `numf`-by-`ndim`, `numf` being the number of facets in the convex hull, and `ndim` the dimension of the space where the points reside.
`AV`	The area or volume of the convex hull.

Definitions

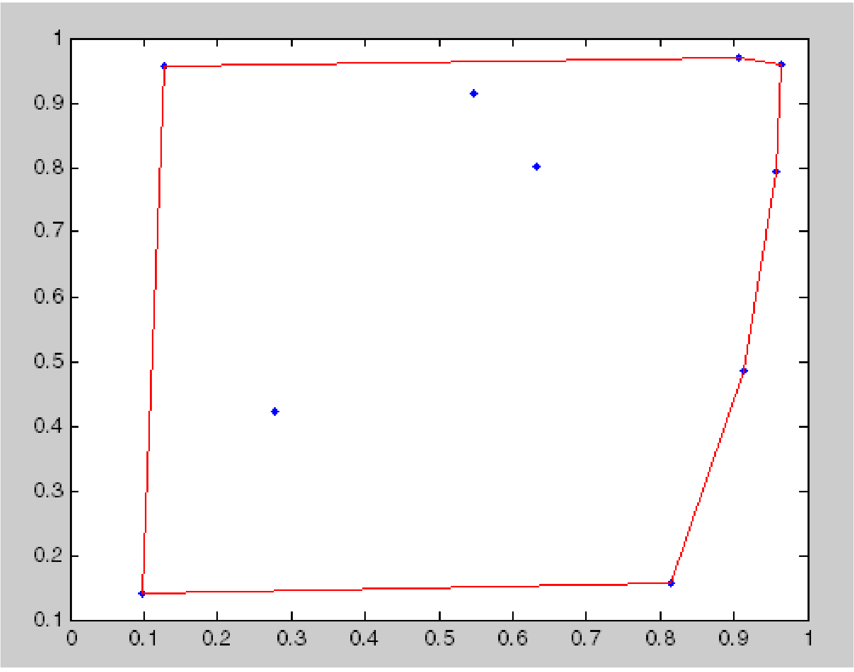
The convex hull of a set of points `X` is the smallest convex region containing all of the points of `X`.

Examples

Example 1

Compute the convex hull of a set of random points located within a unit square in 2-D space.

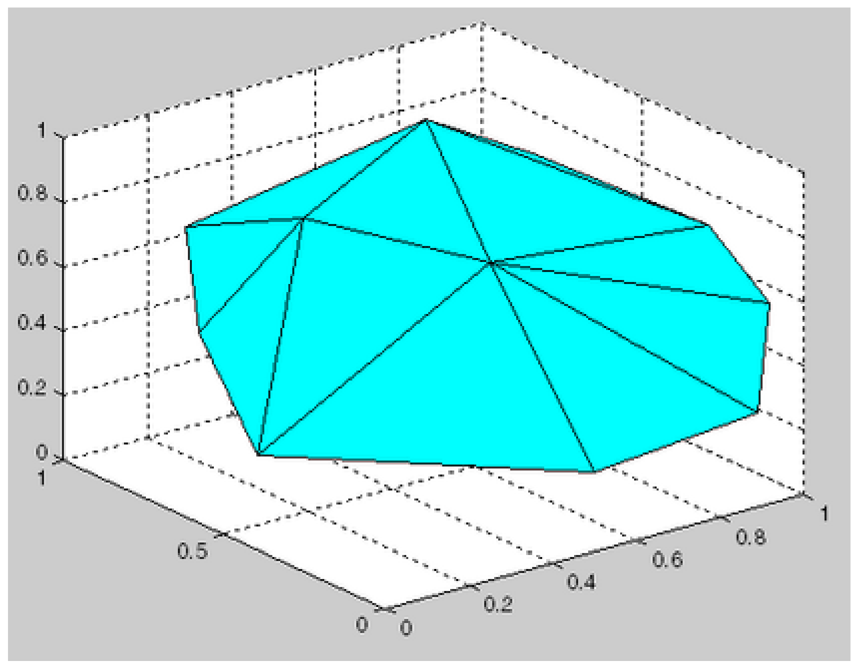
```
x = rand(10,1)
y = rand(10,1)
dt = DelaunayTri(x,y)
k = convexHull(dt)
plot(dt.X(:,1),dt.X(:,2), '.', 'markersize',10); hold on;
plot(dt.X(k,1),dt.X(k,2), 'r'); hold off;
```



Example 2

Compute the convex hull of a set of random points located within a unit cube in 3-D space, and the volume bounded by the convex hull.

```
X = rand(25,3)
dt = DelaunayTri(X)
[ch v] = convexHull(dt)
trisurf(ch, dt.X(:,1),dt.X(:,2),dt.X(:,3), 'FaceColor', 'cyan')
```



See Also

voronoiDiagram | delaunayTriangulation | triangulation | convhull | convhulln

Purpose

Change or view current folder on FTP server

Syntax

```
cd(ftpobj,folder)
cd(ftpobj)
```

Description

`cd(ftpobj,folder)` changes the current folder on the FTP server.

`cd(ftpobj)` displays the current folder on the server.

Input Arguments

ftpobj

FTP object created by `ftp`.

folder

String enclosed in single quotation marks that specifies the target folder. To specify the folder above the current one, use `'..'`.

Examples

Connect to the MathWorks FTP server, change to the `pub` folder, and view its contents:

```
mw=ftp('ftp.mathworks.com');
cd(mw,'pub');
dir(mw)
```

See Also

`dir` | `ftp`

Purpose

Convert complex diagonal form to real block diagonal form

Syntax

```
[V,D] = cdf2rdf(V,D)
[V,D] = cdf2rdf(V,D)
```

Description

If the eigensystem `[V,D] = eig(X)` has complex eigenvalues appearing in complex-conjugate pairs, `cdf2rdf` transforms the system so `D` is in real diagonal form, with 2-by-2 real blocks along the diagonal replacing the complex pairs originally there. The eigenvectors are transformed so that

```
X = V*D/V
```

continues to hold. The individual columns of `V` are no longer eigenvectors, but each pair of vectors associated with a 2-by-2 block in `D` spans the corresponding invariant vectors.

Examples

The matrix

```
X =
    1      2      3
    0      4      5
    0     -5      4
```

has a pair of complex eigenvalues.

```
[V,D] = eig(X)

V =

    1.0000      -0.0191 - 0.4002i     -0.0191 + 0.4002i
         0            0 - 0.6479i           0 + 0.6479i
         0       0.6479                0.6479

D =

    1.0000            0                     0
```

```
         0        4.0000 + 5.0000i          0
         0             0                4.0000 - 5.0000i
```

Converting this to real block diagonal form produces

```
[V,D] = cdf2rdf(V,D)

V =

    1.0000   -0.0191   -0.4002
         0         0   -0.6479
         0    0.6479         0

D =

    1.0000         0         0
         0    4.0000    5.0000
         0   -5.0000    4.0000
```

Algorithms

The real diagonal form for the eigenvalues is obtained from the complex form using a specially constructed similarity transformation.

See Also

`eig` | `rsf2csf`

Purpose

Convert MATLAB formatted dates to CDF formatted dates

Syntax

```
E = cdfepoch(date)
```

Description

`E = cdfepoch(date)` converts the date, specified by `date`, into a `cdfepoch` object. `date` must be a valid date string, returned by `datestr`, or a serial date number, returned by `datenum`. `date` can also be a `cdfepoch` object.

When writing data to a CDF file using `cdfwrite`, use `cdfepoch` to convert MATLAB formatted dates to CDF formatted dates. The MATLAB `cdfepoch` object simulates the `CDFEPOCH` data type in CDF files.

To convert a `cdfepoch` object into a MATLAB serial date number, use the `todatenum` function.

Definitions

The MATLAB serial date number calculates dates differently than CDF epochs.

A MATLAB serial date number represents the whole and fractional number of days from 1-Jan-0000 to a specific date. The year 0000 is merely a reference point and is not intended to be interpreted as a real year in time.

A CDF epoch is the number of milliseconds since 1-Jan-0000.

Examples

Convert the current time in serial date number format into a CDF epoch object.

```
% NOW function returns current time as serial date number
dateobj = cdfepoch(now)

dateobj =

     cdfepoch object:
     11-Mar-2009 15:09:25
```

Convert the current time in date string format into a CDF epoch object.

```
% DATESTR function returns date as string
dateobj2 = cdfepoch(datestr(now))

dateobj2 =

     cdfepoch object:
     11-Mar-2009 15:09:25
```

Convert the CDF epoch object into a serial date number.

```
dateobj = cdfepoch(now);
mydatenum = todatenum(dateobj)

mydatenum =

     7.3384e+005
```

See Also

`datenum` | `datestr` | `todatenum` | `cdfinfo` | `cdfread`

Purpose

Information about Common Data Format (CDF) file

Syntax

```
info = cdfinfo(filename)
```

Description

`info = cdfinfo(filename)` returns information about the Common Data Format (CDF) file specified in the string `filename`.

Note Because `cdfinfo` creates temporary files, the current working directory must be writeable.

The following table lists the fields returned in the structure, `info`. The table lists the fields in the order that they appear in the structure.

Field	Description
`Filename`	Text string specifying the name of the file
`FileModDate`	Text string indicating the date the file was last modified
`FileSize`	Double scalar specifying the size of the file, in bytes
`Format`	Text string specifying the file format
`FormatVersion`	Text string specifying the version of the CDF library used to create the file
`FileSettings`	Structure array containing library settings used to create the file
`Subfiles`	Filenames containing the CDF file's data, if it is a multi-file format CDF

<table>
<tr><th>Field</th><th colspan="2">Description</th></tr>
<tr><td rowspan="6"><code>Variables</code></td><td colspan="2">N-by-6 cell array, where N is the number of variables, containing information about the variables in the file. The columns present the following information:</td></tr>
<tr><td>Column 1</td><td>Text string specifying name of variable</td></tr>
<tr><td>Column 2</td><td>Double array specifying the dimensions of the variable, as returned by the <code>size</code> function</td></tr>
<tr><td>Column 3</td><td>Double scalar specifying the number of records assigned for the variable</td></tr>
<tr><td>Column 4</td><td>Text string specifying the data type of the variable, as stored in the CDF file</td></tr>
<tr><td>Column 5</td><td>Text string specifying the record and dimension variance settings for the variable. The single <code>T</code> or <code>F</code> to the left of the slash designates whether values vary by record. The zero or more <code>T</code> or <code>F</code> letters to the right of the slash designate whether values vary at each dimension. Here are some examples.

<code>T/ (scalar variable</code>
<code>F/T (one-dimensional variable)</code>

T/TFF (three-dimensional variable)</td></tr>
</table>

Field	Description
`GlobalAttributes`	Structure array that contains one field for each global attribute. The name of each field corresponds to the name of an attribute. The data in each field, contained in a cell array, represents the entry values for that attribute.
`VariableAttributes`	Structure array that contains one field for each variable attribute. The name of each field corresponds to the name of an attribute. The data in each field is contained in a *n*-by-2 cell array, where *n* is the number of variables. The first column of this cell array contains the variable names associated with the entries. The second column contains the entry values.

Note Attribute names returned by `cdfinfo` might not match the names of the attributes in the CDF file exactly. Attribute names can contain characters that are illegal in MATLAB field names. `cdfinfo` removes illegal characters that appear at the beginning of attributes and replaces other illegal characters with underscores ('_'). When `cdfinfo` modifies an attribute name, it appends the attribute's internal number to the end of the field name. For example, the attribute name `Variable%Attribute` becomes `Variable_Attribute_013`.

Note To improve performance, turn off the file validation which the CDF library does by default when opening files. For more information, see `cdflib.setValidate`.

Examples

```
info = cdfinfo('example.cdf')
info =
              Filename: 'example.cdf'
```

```
             FileModDate: '09-Mar-2001 15:45:22'
                FileSize: 1240
                  Format: 'CDF'
           FormatVersion: '2.7.0'
            FileSettings: [1x1 struct]
                Subfiles: {}
               Variables: {5x6 cell}
        GlobalAttributes: [1x1 struct]
      VariableAttributes: [1x1 struct]

info.Variables
ans =
  'Time'         [1x2 double] [24] 'epoch'  'T/'     'Full'
  'Longitude'    [1x2 double] [ 1] 'int8'   'F/FT'   'Full'
  'Latitude'     [1x2 double] [ 1] 'int8'   'F/TF'   'Full'
  'Data'         [1x3 double] [ 1] 'double' 'T/TTT'  'Full'
  'multidim'     [1x4 double] [ 1] 'uint8'  'T/TTTT' 'Full'
```

See Also

cdflib.setValidate | cdfread

Purpose

Summary of Common Data Format (CDF) capabilities

Description

The MATLAB product provides both high-level and low-level access to Common Data Format (CDF) files. The high-level access functions make it easy to read data from a CDF file or write data from the workspace to a CDF file. The low-level interface provides you with more control over the import and export operations.

Note For information about MATLAB support for the Network Common Data Form (netCDF), which is a completely separate, incompatible format, see `netcdf`.

High-Level Functions

MATLAB includes functions that provide high-level access to CDF files:

- `cdfinfo`
- `cdfread`

In addition, MATLAB provides functions that convert time data to and from the CDF epoch data type: `cdfepoch` and `todatenum`.

Low-Level Functions

MATLAB provides direct access to dozens of functions in the CDF library. Using these functions, you can read and write data, create variables, attributes, and entries, and take advantage of other features of the CDF library. To use these functions, you must be familiar with the CDF C interface. Documentation about CDF, version 3.3.0, is available at the `CDF Web site`.

The MATLAB functions correspond to functions in the CDF library new Standard Interface. In most cases, the syntax of a MATLAB function is similar to the syntax of the corresponding CDF library function. To use these functions, you must prefix the function name with the package name, `cdflib`. For example, to use the CDF library function to open an existing CDF file, you would use this syntax:

```
cdfid = cdflib.open('example.cdf');
```

The following tables list all of the functions in the MATLAB CDF library package, grouped by category.

Library Information

cdflib	Summary of Common Data Format (CDF) capabilities
cdflib.getConstantNames	Names of Common Data Format (CDF) library constants
cdflib.getConstantValue	Numeric value corresponding to Common Data Format (CDF) library constant
cdflib.getFileBackward	Return current backward compatibility mode setting
cdflib.getLibraryCopyright	Copyright notice of Common Data Format (CDF) library
cdflib.getLibraryVersion	Library version and release information
cdflib.getValidate	Library validation mode
cdflib.setFileBackward	Set backward compatibility mode
cdflib.setValidate	Specify library validation mode

File Operations

cdflib.close	Close Common Data Format (CDF) file
cdflib.create	Create Common Data Format (CDF) file
cdflib.delete	Delete existing Common Data Format (CDF) file
cdflib.getCacheSize	Number of cache buffers used
cdflib.getChecksum	Checksum mode

cdflib.getCompression	Compression settings
cdflib.getCompressionCacheSize	Number of compression cache buffers
cdflib.getCopyright	Copyright notice in Common Data Format (CDF) file
cdflib.getFormat	Format of Common Data Format (CDF) file
cdflib.getMajority	Majority of variables
cdflib.getName	Name of Common Data Format (CDF) file
cdflib.getReadOnlyMode	Read-only mode
cdflib.getStageCacheSize	Number of cache buffers for staging
cdflib.getVersion	Common Data Format (CDF) library version and release information
cdflib.inquire	Basic characteristics of Common Data Format (CDF) file
cdflib.open	Open existing Common Data Format (CDF) file
cdflib.setCacheSize	Specify number of dotCDF cache buffers
cdflib.setChecksum	Specify checksum mode
cdflib.setCompression	Specify compression settings
cdflib.setCompressionCacheSize	Specify number of compression cache buffers
cdflib.setFormat	Specify format of Common Data Format (CDF) file
cdflib.setMajority	Specify majority of variables

cdflib.setReadOnlyMode	Specify read-only mode
cdflib.setStageCacheSize	Specify number of staging cache buffers for Common Data Format (CDF) file

Variables

cdflib.closeVar	Close specified variable from multifile format Common Data Format (CDF) file
cdflib.createVar	Create new variable
cdflib.deleteVar	Delete variable
cdflib.deleteVarRecords	Delete range of records from variable
cdflib.getVarAllocRecords	Number of records allocated for variable
cdflib.getVarBlockingFactor	Blocking factor for variable
cdflib.getVarCacheSize	Number of multifile cache buffers
cdflib.getVarCompression	Information about compression used by variable
cdflib.getVarData	Single value from record in variable
cdflib.getVarMaxAllocRecNum	Maximum allocated record number for variable
cdflib.getVarMaxWrittenRecNum	Maximum written record number for variable
cdflib.getVarName	Variable name, given variable number
cdflib.getVarNum	Variable number, given variable name

cdflib.getVarNumRecsWritten	Number of records written to variable
cdflib.getVarPadValue	Pad value for variable
cdflib.getVarRecordData	Entire record for variable
cdflib.getVarReservePercent	Compression reserve percentage for variable
cdflib.getVarsMaxWrittenRecNum	Maximum written record number for CDF file
cdflib.getVarSparseRecords	Information about how variable handles sparse records
cdflib.hyperGetVarData	Read hyperslab of data from variable
cdflib.hyperPutVarData	Write hyperslab of data to variable
cdflib.inquireVar	Information about variable
cdflib.putVarData	Write single value to variable
cdflib.putVarRecordData	Write entire record to variable
cdflib.renameVar	Rename existing variable
cdflib.setVarAllocBlockRecords	Specify range of records to be allocated for variable
cdflib.setVarBlockingFactor	Specify blocking factor for variable
cdflib.setVarCacheSize	Specify number of multi-file cache buffers for variable
cdflib.setVarCompression	Specify compression settings used with variable
cdflib.setVarInitialRecs	Specify initial number of records written to variable

cdflib.setVarPadValue	Specify pad value used with variable
cdflib.SetVarReservePercent	Specify reserve percentage for variable
cdflib.setVarsCacheSize	Specify number of cache buffers used for all variables
cdflib.setVarSparseRecords	Specify how variable handles sparse records

Attributes

cdflib.createAttr	Create attribute
cdflib.deleteAttr	Delete attribute
cdflib.deleteAttrEntry	Delete attribute entry
cdflib.deleteAttrgEntry	Delete entry in global attribute
cdflib.getAttrEntry	Value of entry in attribute with variable scope
cdflib.getAttrgEntry	Value of entry in global attribute
cdflib.getAttrMaxEntry	Number of last entry for variable attribute
cdflib.getAttrMaxgEntry	Number of last entry for global attribute
cdflib.getAttrName	Name of attribute, given attribute number
cdflib.getAttrNum	Attribute number, given attribute name
cdflib.getAttrScope	Scope of attribute
cdflib.getNumAttrEntries	Number of entries for attribute with variable scope
cdflib.getNumAttrgEntries	Number of entries for attribute with global scope

cdflib.getNumAttributes	Number of attributes with variable scope
cdflib.getNumgAttributes	Number of attributes with global scope
cdflib.inquireAttr	Information about attribute
cdflib.inquireAttrEntry	Information about entry in attribute with variable scope
cdflib.inquireAttrgEntry	Information about entry in attribute with global scope
cdflib.putAttrEntry	Write value to entry in attribute with variable scope
cdflib.putAttrgEntry	Write value to entry in attribute with global scope
cdflib.renameAttr	Rename existing attribute

Utility Functions

cdflib.computeEpoch	Convert time value to `CDF_EPOCH` value
cdflib.computeEpoch16	Convert time value to `CDF_EPOCH16` value
cdflib.epoch16Breakdown	Convert `CDF_EPOCH16` value to time value
cdflib.epochBreakdown	Convert `CDF_EPOCH` value into time value

See Also

`cdfread | cdfinfo`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.close

Purpose

Close Common Data Format (CDF) file

Syntax

`cdflib.close(cdfId)`

Description

`cdflib.close(cdfId)` closes the specified CDF file. `cdfId` identifies the CDF file.

You must close a CDF to guarantee that all modifications you made since opening the CDF are actually written to the file.

Examples

Open the example CDF file and then close it.

```
cdfid = cdflib.open('example.cdf');
cdflib.close(cdfid)
```

References

This function corresponds to the CDF library C API routine `CDFcloseCDF`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.open` | `cdflib.create`

Tutorials

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

Purpose

Close specified variable from multifile format Common Data Format (CDF) file

Syntax

```
cdflib.closeVar(cdfId,varNum)
```

Description

`cdflib.closeVar(cdfId,varNum)` closes a variable in a multifile format CDF.

`cdfId` identifies the CDF file and `varNum` is a numeric value that specifies the variable. Variable identifiers (variable numbers) are zero-based.

For multifile CDFs, you must close all open variable files to guarantee that all modifications you have made are actually written to the CDF file(s). You do not need to call this function for variables in a single-file format CDF.

Examples

Create a multifile CDF, create a variable, and then close the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_multifile.cdf');

% Make it a multifile format CDF
cdflib.setFormat(cdfid,'MULTI_FILE')

% Create a variable in the CDF.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

%  Close the variable.
cdflib.closeVar(cdfid, varnum)

% Clean up
cdflib.delete(cdfid)
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFclosezVar`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarNum` | `cdflib.setFormat` | `cdflib.getFormat`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Convert time value to CDF_EPOCH value

Syntax

```
epoch = cdflib.computeEpoch(timeval)
```

Description

`epoch = cdflib.computeEpoch(timeval)` converts the time value specified by `timeval` into a `CDF_EPOCH` value.

Input Arguments

timeval

7-by-1 time vector. The following table describes the time components.

Component	Description
year	AD e.g. 1994
month	1 12
day	1 31
hour	0 23
minute	0 59
second	0 59
millisecond	0 999

Output Arguments

epoch

MATLAB `double` representing a `CDF_EPOCH` time value.

Examples

Convert a time value into a `CDF_EPOCH` value.

```
timeval = [1999 12 31 23 59 59 0];
epoch = cdflib.computeEpoch(timeval);
```

References

This function corresponds to the CDF library C API routine `computeEPOCH`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.computeEpoch16` | `cdflib.epochBreakdown` | `cdflib.epoch16Breakdown`

Tutorials

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

Purpose Convert time value to `CDF_EPOCH16` value

Syntax

```
epoch16 = cdflib.computeEpoch16(timeval)
```

Description `epoch16 = cdflib.computeEpoch16(timeval)` converts the time value specified by `timeval` into a `CDF_EPOCH16` value.

Input Arguments

timeval

10-by-1 time vector. The following table describes the time components. To specify multiple time values, use additional columns.

Component	Description
year	AD e.g. 1994
month	1 12
day	1 31
hour	0 23
minute	0 59
second	0 59
millisecond	0 999
microsecond	0 999
nanosecond	0 999
picosecond	0 999

Output Arguments

epoch16

CDF Epoch16 time value. If the input argument `timeval` has `m-by-10` elements, the return value `epoch16` will have size `2-by-m`

Examples Convert the time value into an `CDF_EPOCH16` value:

```
timeval = [1999; 12; 31; 23; 59; 59; 50; 100; 500; 999];
```

```
epoch16 = cdflib.computeEpoch16(timeval);
```

References

This function corresponds to the CDF library C API routine `computeEPOCH16`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.computeEpoch` | `cdflib.epochBreakdown` | `cdflib.epoch16Breakdown`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Create Common Data Format (CDF) file

Syntax

`cdfId = cdflib.create(filename)`

Description

`cdfId = cdflib.create(filename)` creates a new CDF file with the name specified by the text string *`filename`*. Returns the CDF file identifier `cdfId`.

Examples

Create a CDF file. To run this example, you must have write permission in your current directory.

```
cdfId = cdflib.create('myfile.cdf');

% Clean up
cdflib.delete(cdfId);

clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFcreateCDF`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.open` | `cdflib.close`

Tutorials

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

Purpose

Create attribute

Syntax

```
attrnum = cdflib.createAttr(cdfId,attrName,scope)
```

Description

`attrnum = cdflib.createAttr(cdfId,attrName,scope)` creates an attribute in a CDF file with the specified scope.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

attrName

Text string that specifies the name you want to assign to the attribute.

scope

One of the following text strings, or its numeric equivalent, that specifies the scope of the attribute.

Text String	Description
`'global_scope'`	Attribute applies to the CDF as a whole.
`'variable_scope'`	Attribute applies only to the variable

To get the numeric equivalent of these text string constants, use the `cdflib.getConstantValue` function.

Output Arguments

attrNum

Numeric value identifying the attribute. Attribute numbers are zero-based.

Examples

Create a CDF, and then create an attribute in the CDF. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create attribute
attrNum = cdflib.createAttr(cdfId,'Purpose','global_scope');

% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFcreateAttr`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrNum` | `cdflib.deleteAttr` | `cdflib.getConstantValue` | `cdflib.getConstantNames`

Tutorials

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

cdflib.createVar

Purpose Create new variable

Syntax

```
varnum = cdflib.createVar(cdfId, varname, datatype,
   numElements, dims, recVariance, dimVariance)
```

Description

`varnum = cdflib.createVar(cdfId, varname, datatype, numElements, dims, recVariance, dimVariance)` creates a new variable in the Common Data Format (CDF) file with the specified characteristics.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varname

Text string that specifies the name you want to assign to the variable.

datatype

Data type of the variable. One of the following text strings, or its numeric equivalent, that specifies a valid CDF data type.

CDF Data Type	Description
`CDF_BYTE`	1-byte, signed integer
`CDF_CHAR`	1 byte, signed character data type that maps to the MATLAB `char` class
`CDF_INT1`	1-byte, signed integer
`CDF_UCHAR`	1 byte, unsigned character data type that maps to the MATLAB `uint8` class
`CDF_UINT1`	1-byte, unsigned integer
`CDF_INT2`	2-byte, signed integer
`CDF_UINT2`	2-byte, unsigned integer

CDF Data Type	Description
CDF_INT4	4-byte, signed integer
CDF_UINT4	4-byte, unsigned integer
CDF_FLOAT	4-byte, floating point
CDF_REAL4	4-byte, floating point
CDF_REAL8	8-byte, floating point.
CDF_DOUBLE	8-byte, floating point
CDF_EPOCH	8-byte, floating point
CDF_EPOCH16	two 8-byte, floating point

numElements

Number of elements per datum. Value should be 1 for all data types, except `CDF_CHAR` and `CDF_UCHAR`.

dims

A vector of the dimensions extents; empty if there are no dimension extents.

recVariance

Specifies record variance: `true` or `false`.

dimVariance

A vector of logicals; empty if there are no dimensions.

Output Arguments

varNum

The numeric identifier for the variable. Variable numbers are zero-based.

Examples

Create a CDF file and then create a variable named `'Time'` in the CDF. The variable has no dimensions and varies across records. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Initially the file contains no variables.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 0
    numvAttrs: 0
    numgAttrs: 0

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Retrieve info about the file again to verify variable was created.
% Note value of numVars field is now 1.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 1
    numvAttrs: 0
    numgAttrs: 0

% Clean up
cdflib.delete(cdfid);
```

```
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFcreatezVar`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.deleteVar` | `cdflib.closeVar` |

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.delete

Purpose

Delete existing Common Data Format (CDF) file

Syntax

```
cdflib.delete(cdfId)
```

Description

`cdflib.delete(cdfId)` deletes the existing CDF file specified by the identifier `cdfId`. If the CDF file is a multi-file format CDF, the `cdflib.delete` function also deletes the variable files (having file extensions of `.z0`, `.z1`, etc.).

Examples

Create a CDF file, and then delete it. To run this example, you must be in a writable folder.

```
cdfId = cdflib.create('mytempfile.cdf');

% Verify that the file was created.
ls *.cdf

mytempfile.cdf

% Delete the file.
cdflib.delete(cdfId)

% Verify that the file no longer exists.
ls *.cdf
```

References

This function corresponds to the CDF library C API routine `CDFdeleteCDF`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.create` | `cdflib.setFormat`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Delete attribute

Syntax

`cdflib.deleteAttr(cdfId,attrNum)`

Description

`cdflib.deleteAttr(cdfId,attrNum)` deletes the specified attribute from the CDF file.

`cdfId` identifies the Common Data Format (CDF) file.`attrNum` is a numeric identifier that specifies the attribute. Attribute numbers are zero-based.

Examples

Create a CDF file, and then create an attribute in the file. Then delete the attribute. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create attribute.
attrNum = cdflib.createAttr(cdfId,'Purpose','global_scope');

% Prove it exists.
anum = cdflib.getAttrNum(cdfid,'Purpose')

anum =

     0

% Delete the attribute.
cdflib.deleteAttr(cdfid,attrNum);


% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFdeleteAttr`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createAttr` | `cdflib.getAttrNum`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Delete attribute entry

Syntax

`cdflib.deleteAttrEntry(cdfId,attrNum,entryNum)`

Description

`cdflib.deleteAttrEntry(cdfId,attrNum,entryNum)` deletes an entry from an attribute in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

attrNum

Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have variable scope.

entryNum

Numeric value that specifies the entry in the attribute. Entry numbers are zero-based.

Examples

Create a CDF, and then create an attribute in the file. Write a value to an entry for the attribute, and then delete the entry. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Initially the file contains no attributes, global or variable.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 0
    numvAttrs: 0
```

```
    numgAttrs: 0

% Create an attribute with variable scope in the file.
attrNum = cdflib.createAttr(cdfid,'my_var_scope_attr','variable_scope');

% Write a value to an entry for the attribute
cdflib.putAttrEntry(cdfid,attrNum,0,'CDF_CHAR','My attr value');

% Get the value of the attribute entry
value = cdflib.getAttrEntry(cdfid,attrNum,0)

value =

My attr value

% Delete the entry
cdflib.deleteAttrEntry(cdfid,attrNum,0);

% Now try to view the value of the entry
% Should return NO_SUCH_ENTRY failure.
value = cdflib.getAttrEntry(cdfid,attrNum,0)  % Should fail

% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFdeleteAttrzEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.deleteAttr`

Tutorials

- “Importing Common Data File Format (CDF) Files”

- “Exporting to Common Data File Format (CDF) Files”

cdflib.deleteAttrgEntry

Purpose Delete entry in global attribute

Syntax `cdflib.deleteAttrgEntry(cdfId,attrNum,entryNum)`

Description `cdflib.deleteAttrgEntry(cdfId,attrNum,entryNum)` deletes an entry from a global attribute in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

attrNum

Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have global scope.

entryNum

Numeric value that specifies the entry in the attribute. Entry numbers are zero-based.

Examples Create a CDF and create a global attribute in the file. Write a value to an entry for the attribute and then delete the entry. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Initially the file contains no attributes, global or variable.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 0
    numvAttrs: 0
```

```
    numgAttrs: 0

% Create an attribute with global scope in the file.
attrNum = cdflib.createAttr(cdfid,'my_global_attr','global_scope');

% Write a value to an entry for the attribute
cdflib.putAttrgEntry(cdfid,attrNum,0,'CDF_CHAR','My global attr');

% Get the value of the global attribute entry
value = cdflib.getAttrgEntry(cdfid,attrNum,0)

value =

My global attr

% Delete the entry
cdflib.deleteAttrgEntry(cdfid,attrNum,0);

% Now try to view the value of the entry
% Should return NO_SUCH_ENTRY failure.
value = cdflib.getAttrgEntry(cdfid,attrNum,0)  % Should fail

% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFdeleteAttrgEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.deleteAttr` | `cdflib.deleteAttrEntry`

Tutorials

- “Importing Common Data File Format (CDF) Files”

- "Exporting to Common Data File Format (CDF) Files"

Purpose Delete variable

Syntax

```
cdflib.deleteVar(cdfId,varNum)
```

Description

`cdflib.deleteVar(cdfId,varNum)` deletes a variable from a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that specifies the variable. Variable numbers are zero-based.

Examples

Create a CDF, create a variable in the CDF, and then delete it.

```
cdfid = cdflib.create('mycdf.cdf');

% Initially the file contains no variables.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 0
    numvAttrs: 0
    numgAttrs: 0

% Create a variable in the CDF.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Retrieve info about the variable in the CDF.

info = cdflib.inquireVar(cdfid, 0)

info =

           name: 'Time'
       datatype: 'cdf_int1'
```

```
    numElements: 1
           dims: []
    recVariance: 1
    dimVariance: []

% Delete the variable from the CDF

cdflib.deleteVar(cdfid,0);

% Check to see if the variable was deleted from the CDF.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 0
    numvAttrs: 0
    numgAttrs: 0

% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFdeletezVar`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createVar`

Tutorials

- "Importing Common Data File Format (CDF) Files"

- “Exporting to Common Data File Format (CDF) Files”

cdflib.deleteVarRecords

Purpose

Delete range of records from variable

Syntax

`cdflib.deleteVarRecords(cdfId,varNum,startRec,endRec)`

Description

`cdflib.deleteVarRecords(cdfId,varNum,startRec,endRec)` deletes a range of records from a variable in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varNum

Numeric value that identifies the variable. Variable numbers are zero-based.

startRec

Numeric value that specifies the record at which to start deleting records. Record numbers are zero-based.

endRec

Numeric value that specifies the record at which to stop deleting records. Record numbers are zero-based.

Examples

Make a writable copy of the example CDF, get the number of a variable in the CDF, and delete specific records in the variable. To run this example, you must be in a writable folder.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.cdf');
copyfile(srcFile,'myfile.cdf');
fileattrib('myfile.cdf','+w');
cdfid = cdflib.open('myfile.cdf');
varnum = cdflib.getVarNum(cdfid,'Temperature');
cdflib.deleteVarRecords(cdfid,varnum,1,2);
cdflib.close(cdfid);
```

References

This function corresponds to the CDF library C API routine `CDFdeletezVarRecords`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarNumRecsWritten` | `cdflib.putVarRecordData`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.epoch16Breakdown

Purpose

Convert `CDF_EPOCH16` value to time value

Syntax

```
timeVec = cdflib.epoch16Breakdown(epoch16Time)
```

Description

`timeVec = cdflib.epoch16Breakdown(epoch16Time)` convert a `CDF_EPOCH16` value into a time vector. `timeVec` will have 10-by-*n* elements, where *n* is the number of CDF_EPOCH16 values.

The following table describes the time value components.

timeVec Element	Description	Valid Values
`timeVec(1,:)`	Year AD	e.g. `1994`
`timeVec(2,:)`	Month	`1 12`
`timeVec(3,:)`	Day	`1 31`
`timeVec(4,:)`	Hour	`0 23`
`timeVec(5,:)`	Minute	`0 59`
`timeVec(6,:)`	Second	`0 59`
`timeVec(7,:)`	Millisecond	`0 999`
`timeVec(8,:)`	Microsecond	`0 999`
`timeVec(9,:)`	Nanosecond	`0 999`
`timeVec(10,:)`	Picosecond	`0 999`

Examples

Convert `CDF_EPOCH16` value into time value.

```
timeval = [1999; 12; 31; 23; 59; 59; 50; 100; 500; 999];
epoch16 = cdflib.computeEpoch16(timeval);

timevec = cdflib.epoch16Breakdown(epoch16)

timevec =

        1999
```

```
    12
    31
    23
    59
    59
    50
   100
   500
   999
```

References

This function corresponds to the CDF library C API routine `EPOCH16breakdown`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.computeEpoch16`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.epochBreakdown

Purpose

Convert CDF_EPOCH value into time value

Syntax

```
timeVec = cdflib.epochBreakdown(epochTime)
```

Description

`timeVec = cdflib.epochBreakdown(epochTime)` decomposes the `CDF_EPOCH` value, `epochTime` value into individual time components. `timeVec` will have 7-by-*n* elements, where *n* is the number of `CDF_EPOCH` values in `epochTime`.

The return value `timeVec` has the following elements:

timeVec Element	Description	Valid Values
`timeVec(1,:)`	Year AD	e.g. `1994`
`timeVec(2,:)`	Month	`1 12`
`timeVec(3,:)`	Day	`1 31`
`timeVec(4,:)`	Hour	`0 23`
`timeVec(5,:)`	Minute	`0 59`
`timeVec(6,:)`	Second	`0 59`
`timeVec(7,:)`	Millisecond	`0 999`

Examples

Convert a `CDP_EPOCH` value into a time vector.

```
% First convert a time vector into a CDF_EPOCH value
timeval = [1999 12 31 23 59 59 0];
epoch = cdflib.computeEpoch(timeval);

% Convert the CDF_EPOCH value into a time vector
timevec = cdflib.epochBreakdown(epoch)

timevec =

        1999
          12
          31
```

```
23
59
59
 0
```

References

This function corresponds to the CDF library C API routine `EPOCHbreakdown`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.computeEpoch` | `cdflib.epochBreakdown` | `cdflib.epoch16Breakdown`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.getAttrEntry

Purpose

Value of entry in attribute with variable scope

Syntax

```
value = cdflib.getAttrEntry(cdfId,attrNum,entryNum)
```

Description

`value = cdflib.getAttrEntry(cdfId,attrNum,entryNum)` returns the value of an attribute entry in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

attrNum

Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have variable scope.

entryNum

Numeric value that specifies the entry in the attribute. Entry numbers are zero-based.

Output Arguments

Value

Value of the entry.

Examples

Open the example CDF and get the value of an entry associated with an attribute with variable scope in the file.

```
cdfid = cdflib.open('example.cdf');

% The fourth attribute is of variable scope.
attrscope = cdflib.getAttrScope(cdfid,3)

attrscope =

VARIABLE_SCOPE
```

```
% Get information about the first entry for this attribute
[dtype numel] = cdflib.inquireAttrEntry(cdfid,3,0)

dtype =

cdf_char


numel =

    10

% Get the value of the entry for this attribute.
% Note that it's a character string, 10 characters in length
value = cdflib.getAttrEntry(cdfid,3,0)

value =

Time value

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrzEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.putAttrEntry` | `cdflib.getAttrgEntry` | `cdflib.putAttrgEntry`

Tutorials

- "Importing Common Data File Format (CDF) Files"

- "Exporting to Common Data File Format (CDF) Files"

Purpose

Value of entry in global attribute

Syntax

```
value = cdflib.getAttrgEntry(cdfId,attrNum,entryNum)
```

Description

`value = cdflib.getAttrgEntry(cdfId,attrNum,entryNum)` returns the value of a global attribute entry in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

attrNum

Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have global scope.

entryNum

Numeric value that specifies the entry in the attribute. Entry numbers are zero-based.

Output Arguments

Value

Value of the entry.

Examples

Open the example CDF, and then get the value of an entry associated with a global attribute in the file:

```
cdfid = cdflib.open('example.cdf');

% Any of the first three attributes have global scope.
attrscope = cdflib.getAttrScope(cdfid,0)

attrscope =

GLOBAL_SCOPE
```

```
% Get information about the first entry for global attribute
[dtype numel] = cdflib.inquireAttrgEntry(cdfid,0,0)

dtype =

cdf_char


numel =

    23

% Get the value of the first entry for this global attribute.
value = cdflib.getAttrgEntry(cdfid,0,0)

value =

This is a sample entry.

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrgEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.putAttrgEntry` | `cdflib.getAttrEntry` | `cdflib.putAttrEntry`

Tutorials

- “Importing Common Data File Format (CDF) Files”

- "Exporting to Common Data File Format (CDF) Files"

cdflib.getAttrMaxEntry

Purpose

Number of last entry for variable attribute

Syntax

```
maxEntry = cdflib.getAttrMaxEntry(cdfId,attrNum)
```

Description

`maxEntry = cdflib.getAttrMaxEntry(cdfId,attrNum)` returns the number of the last entry for an attribute in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file.

`attrNum` is a numeric value that specifies the attribute. Attribute numbers are zero-based. The attribute must have variable scope.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

attrNum

Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have variable scope.

Output Arguments

maxEntry

Entry number of the last entry in the attribute. Entry numbers are zero-based.

Examples

Open the example CDF and get the number of the last entry associated with an attribute with variable scope in the file:

```
cdfid = cdflib.open('example.cdf');

% The fourth attribute is of variable scope.
attrscope = cdflib.getAttrScope(cdfid,3)

attrscope =

VARIABLE_SCOPE
```

```
% Get the number of the last entry for this attribute.
entrynum = cdflib.getAttrMaxEntry(cdfid,3)

entrynum =

     3


% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrMaxzEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrMaxgEntry`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.getAttrMaxgEntry

Purpose

Number of last entry for global attribute

Syntax

```
maxEntry = cdflib.getAttrMaxgEntry(cdfId,attrNum)
```

Description

`maxEntry = cdflib.getAttrMaxgEntry(cdfId,attrNum)` returns the last entry number of a global attribute in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

attrNum

Numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have global scope.

Output Arguments

maxEntry

Entry number of the last entry in the attribute. Entry numbers are zero-based.

Examples

Open the example CDF and get the number of the last entry associated with a global attribute in the file:

```
cdfid = cdflib.open('example.cdf');

% Any of the first three attribute are of global scope.
attrscope = cdflib.getAttrScope(cdfid,0)

attrscope =

GLOBAL_SCOPE


% Get the number of the last entry for this attribute.
entrynum = cdflib.getAttrMaxgEntry(cdfid,0)
```

```
entrynum =

     4

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrMaxgEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrMaxEntry`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.getAttrName

Purpose

Name of attribute, given attribute number

Syntax

```
name = cdflib.getAttrName(cdfId,attrNum)
```

Description

`name = cdflib.getAttrName(cdfId,attrNum)` returns the name of an attribute in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

attrNum

Numeric value that identifies the attribute. Attribute numbers are zero-based.

Output Arguments

name

Text string specifying the name of the attribute.

Examples

Open the example CDF and get name of an attribute.

```
cdfid = cdflib.open('example.cdf');

% Get name of the first attribute in the file.
attrName = cdflib.getAttrName(cdfId,0)

attrName =

SampleAttribute

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrName`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createAttr`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.getAttrNum

Purpose

Attribute number, given attribute name

Syntax

```
attrNum = cdflib.getAttrNum(cdfId,attrName)
```

Description

`attrNum = cdflib.getAttrNum(cdfId,attrName)` returns the number of an attribute in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

attrName

Text string specifying the name of an attribute.

Output Arguments

attrNum

Numeric value that identifies the attribute. Attribute numbers are zero-based.

Examples

Open the example CDF and get the attribute number associated with the `SampleAttribute` attribute.

```
cdfid = cdflib.open('example.cdf');

attrNum = cdflib.getAttrNum(cdfid,'SampleAttribute')

attrNum =

     0

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrNum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createAttr` | `cdflib.getAttrName`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.getAttrScope

Purpose

Scope of attribute

Syntax

```
scope = cdflib.getAttrScope(cdfId,attrNum)
```

Description

`scope = cdflib.getAttrScope(cdfId,attrNum)` returns the scope of an attribute in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

attrNum

Numeric value that specifies the attribute. Attribute numbers are zero-based.

Output Arguments

scope

Either of the following text strings, or its numeric equivalent.

Text String	Description
`'GLOBAL_SCOPE'`	Attribute applies to the CDF as a whole.
`'VARIABLE_SCOPE'`	Attribute applies only to the variable.

To get the numeric equivalent of these text string constants, use the `cdflib.getConstantValue` function.

Examples

Open example CDF and get the scope of the first attribute in the file:

```
cdfid = cdflib.open('example.cdf');

attrScope = cdflib.getAttrScope(cdfid,0)

attrScope =
```

```
GLOBAL_SCOPE

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetAttrScope`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createAttr` | `cdflib.getAttrName` | `cdflib.getConstantValue`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.getCacheSize

Purpose Number of cache buffers used

Syntax

```
numBuffers = cdflib.getCacheSize(cdfId)
```

Description `numBuffers = cdflib.getCacheSize(cdfId)` returns the number of cache buffers used for the Common Data Format (CDF) file identified by `cdfId`. For a discussion of cache schemes, see the *CDF User's Guide*.

Examples Open the example CDF file and get the cache size:

```
cdfid = cdflib.open('example.cdf');

numBuf = cdflib.getCacheSize(cdfid)

numBuf =

   300

% Clean up
cdflib.close(cdfid)
clear cdfid
```

References This function corresponds to the CDF library C API routine `CDFgetCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also `cdflib.setCacheSize`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Checksum mode

Syntax

```
mode = cdflib.getChecksum(cdfId)
```

Description

`mode = cdflib.getChecksum(cdfId)` returns the checksum mode of the Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

mode

Either of the following text strings or its numeric equivalent.

'MD5_CHECKSUM'	File uses MD5 checksum.
'NO_CHECKSUM'	File does not use a checksum.

To get the numeric equivalent of these text strings, use `cdflib.getConstantValue`.

Examples

Open the example CDF file, and then get the checksum mode of the file:

```
cdfid = cdflib.open('example.cdf');

checksummode = cdflib.getChecksum(cdfid)


checksummode =

NO_CHECKSUM

% Clean up
cdflib.close(cdfid);
clear cdfid;
```

References

This function corresponds to the CDF library C API routine `CDFgetChecksum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setChecksum` | `cdflib.getConstantValue`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Compression settings

Syntax

`[ctype,cparms,cpercentage] = cdflib.getCompression(cdfId)`

Description

`[ctype,cparms,cpercentage] = cdflib.getCompression(cdfId)` returns information about the compression settings of a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

ctype

Text string specifying compression type, such as `'HUFF_COMPRESSION'`. If the CDF does not use compression, the function returns the string `'NO_COMPRESSION'`. For a list of supported compression types, see `cdflib.setCompression`.

cparms

The value of the parameter associated with the type of compression. For example, for the `'RLE_COMPRESSION'` compression type, the parameter specifies the style of run-length encoding. For a list of parameters supported by each compression type, see `cdflib.setCompression`.

cpercentage

The rate of compression, expressed as a percentage.

Examples

Open the example CDF file and check the compression settings in the file.

```
cdfId = cdflib.open('example.cdf');

[ctype, cparms, cpercentage] = cdflib.getCompression(cdfId)
```

```
ctype =

GZIP_COMPRESSION

cparms =

     7

cper =

    26

% Clean up
cdflib.close(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetCompression`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setCompression` | `cdflib.getVarCompression` | `cdflib.setVarCompression`

Tutorials

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

Purpose

Number of compression cache buffers

Syntax

```
numBuffers = cdflib.getCompressionCacheSize(cdfId)
```

Description

`numBuffers = cdflib.getCompressionCacheSize(cdfId)` returns the number of cache buffers used for the compression scratch Common Data Format (CDF) file. `cdfId` identifies the CDF file. For a discussion of cache schemes, see the *CDF User's Guide*.

Examples

Open the example CDF file and check the compression cache size of the file:

```
cdfId = cdflib.open('example.cdf');

numBuf = cdflib.getCompressionCacheSize(cdfId)

numBuf =

    80

% Clean up
cdflib.close(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetCompressionCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setCompressionCacheSize`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.getConstantNames

Purpose

Names of Common Data Format (CDF) library constants

Syntax

```
names = cdflib.getConstantNames()
```

Description

`names = cdflib.getConstantNames()` returns a cell array of text strings, where each text string is the name of a constant known to the Common Data Format (CDF) library.

Examples

Get a list of the names of CDF library constants.

```
names  = cdflib.getConstantNames()

names =

    'AHUFF_COMPRESSION'
    'ALPHAMVSD_ENCODING'
    'ALPHAMVSG_ENCODING'
    'ALPHAMVSI_ENCODING'
    'ALPHAOSF1_ENCODING'
    'CDF_BYTE'
    'CDF_CHAR'
      .
      .
      .
```

References

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getConstantValue`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Numeric value corresponding to Common Data Format (CDF) library constant

Syntax

`value = cdflib.getConstantValue(constantName)`

Description

`value = cdflib.getConstantValue(constantName)` returns the numeric value of the CDF library constant specified by the text string *`constantName`*. To see a list of constant names, use `cdflib.getConstantNames`.

Examples

View the list of CDF library constants and get the numeric value corresponding to one of the constants.

```
% Retrieve a list of library constants
names = cdflib.getConstantNames();

value  = cdflib.getConstantValue(names{1})

value =

   3
```

References

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getConstantNames`

Tutorials

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

cdflib.getCopyright

Purpose

Copyright notice in Common Data Format (CDF) file

Syntax

```
copyright = cdflib.getCopyright(cdfId)
```

Description

`copyright = cdflib.getCopyright(cdfId)` returns the copyright notice in the CDF file identified by `cdfId`.

Examples

Create a CDF file, and then get the copyright notice in the file. To run this example, you must be in a writable folder.

```
cdfId = cdflib.create('your_file.cdf');

copyright  = cdflib.getCopyright(cdfId)

copyright =

Common Data Format (CDF)
(C) Copyright 1990-2009 NASA/GSFC
Space Physics Data Facility
NASA/Goddard Space Flight Center
Greenbelt, Maryland 20771 USA
(Internet -- CDFSUPPORT@LISTSERV.GSFC.NASA.GOV)

% Clean up.
cdflib.delete(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetCopyright`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getLibraryCopyright`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.getFileBackward

Purpose

Return current backward compatibility mode setting

Syntax

```
mode = cdflib.getFileBackward()
```

Description

`mode = cdflib.getFileBackward()` returns the backward compatibility mode.

Output Arguments

mode

One of the following text strings:

`BACKWARDFILEon`	Backward compatibility mode is on.
`BACKWARDFILEoff`	Backward compatibility mode is off.

For more information about backward compatibility mode, see `cdflib.setFileBackward`.

Examples

```
mode = cdflib.getFileBackward

mode =

BACKWARDFILEoff
```

References

This function corresponds to the CDF library C API routine `CDFgetFileBackward`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setFileBackward` | `cdflib.getConstantValue`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"
-

Purpose

Format of Common Data Format (CDF) file

Syntax

format = cdflib.setFormat(cdfId)

Description

format = cdflib.setFormat(cdfId) returns the format of the CDF file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to cdflib.create or cdflib.open.

Output Arguments

format

Either of the following text strings, or its numeric equivalent.

'SINGLE_FILE'	The CDF is stored in a single file.
'MULTI_FILE'	The CDF is made up of multiple files.

To get the numeric equivalent of these text strings, use cdflib.getConstantValue.

Examples

Open the example CDF file and determine its file format:

```
cdfId = cdflib.open('example.cdf');

format = cdflib.getFormat(cdfId)

format =

   'SINGLE_FILE'

% Clean up.
cdflib.close(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetFormat`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setFormat` | `cdflib.getConstantValue`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Copyright notice of Common Data Format (CDF) library

Syntax

```
copyright = cdflib.getLibraryCopyright()
```

Description

`copyright = cdflib.getLibraryCopyright()` returns a text string containing the copyright notice of the CDF library.

Examples

Get the copyright of the CDF library.

```
copyright  = cdflib.getLibraryCopyright()

copyright =

Common Data Format (CDF)
(C) Copyright 1990-2008 NASA/GSFC
Space Physics Data Facility
NASA/Goddard Space Flight Center
Greenbelt, Maryland 20771 USA
(Internet -- CDFSUPPORT@LISTSERV.GSFC.NASA.GOV)
```

References

This function corresponds to the CDF library C API routine `CDFgetLibraryCopyright`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getCopyright`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.getLibraryVersion

Purpose

Library version and release information

Syntax

```
[version,release,increment] = cdflib.getLibraryVersion()
```

Description

`[version,release,increment] = cdflib.getLibraryVersion()` returns information about the Common Data Format (CDF) library.

Output Arguments

version

Numeric value indicating the version number of the CDF library.

release

Numeric value indicating the release number of the CDF library.

increment

Numeric value indicating the increment number of the CDF library.

Examples

Get the version information of the CDF library:

```
[version, release, increment] = cdflib.getLibraryVersion()

version =

     3

release =

     3

increment =

     0
```

References

This function corresponds to the CDF library C API routine `CDFgetLibraryVersion`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVersion`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.getMajority

Purpose

Majority of variables

Syntax

```
majority = cdflib.getMajority(cdfId)
```

Description

majority = `cdflib.getMajority(cdfId)` returns the majority of variables in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

majority

Either of the following text strings, or its numeric equivalent.

'ROW_MAJOR'	C-like array ordering for variable storage. The first dimension in each variable array varies the slowest. This is the default.
'COLUMN_MAJOR'	Fortran-like array ordering for variable storage. The first dimension in each variable array varies the fastest.

To get the numeric equivalent of these values, use `cdflib.getConstantValue`.

Examples

Open the example CDF file, and then determine the majority of variables in the file:

```
cdfId = cdflib.open('example.cdf');

majority = cdflib.getMajority(cdfId)

majority =

ROW_MAJOR
```

```
% Clean up
cdflib.close(cdfId)

clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetMajority`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setMajority` | `cdflib.getConstantValue`

Tutorials

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

cdflib.getName

Purpose

Name of Common Data Format (CDF) file

Syntax

```
name = cdflib.getName(cdfId)
```

Description

`name = cdflib.getName(cdfId)` returns the name of the CDF file identified by `cdfId`.

Examples

Open the example CDF file and get the name of the file. The path name returned for your installation will be different.

```
cdfId = cdflib.open('example.cdf');

name = cdflib.getName(cdfId)

name =

yourinstallation\matlab\toolbox\matlab\demos\example

% Clean up
cdflib.close(cdfId)

clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetName`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.open` | `cdflib.create` |

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Number of entries for attribute with variable scope

Syntax

```
nentries = cdflib.getNumAttrEntries(cdfId,attrNum)
```

Description

`nentries = cdflib.getNumAttrEntries(cdfId,attrNum)` returns the number of entries for the specified attribute in the Common Data Format (CDF) file.

`cdfId` identifies the CDF file.

`attrNum` is a numeric value that specifies the attribute. Attribute numbers are zero-based. The attribute must have variable scope.

Examples

Open the example CDF, find an attribute with variable scope, and determine how many entries are associated with the attribute:

```
cdfid = cdflib.open('example.cdf');

% Get the number of an attribute
% with variable scope
attrNum = cdflib.getAttrNum(cdfid,'Description');

% Check that scope of attribute is variable
attrScope = cdflib.getAttrScope(cdfid,attrNum)

VARIABLE_SCOPE

% Detemine the number of entries for the attribute
attrEntries = cdflib.getNumAttrEntries(cdfid,attrNum)

attrEntries =

    4

% Clean up
cdflib.close(cdfid);
```

References

This function corresponds to the CDF library C API routine `CDFgetNumAttrzEntries`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrScope`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.getNumAttrgEntries

Purpose

Number of entries for attribute with global scope

Syntax

```
nentries = cdflib.getNumAttrgEntries(cdfId,attrNum)
```

Description

`nentries = cdflib.getNumAttrgEntries(cdfId,attrNum)` returns the number of entries written for the specified global attribute in the Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `attrNum` is a numeric value that identifies the attribute. Attribute numbers are zero-based. The attribute must have global scope.

Examples

Open the example CDF and find out how many entries are associated with a global attribute in the file.

```
cdfid = cdflib.open('example.cdf');

% The first attribute is a global attribute.
attrgEntries = cdflib.getNumAttrgEntries(cdfid,0)

attrgEntries =

     3

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetNumAttrgEntries`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getNumAttrEntries`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose Number of attributes with variable scope

Syntax

```
numAtts = cdflib.getNumAttributes(cdfId)
```

Description `numAtts = cdflib.getNumAttributes(cdfId)` returns the total number of attributes with variable scope in a Common Data Format (CDF) file. `cdfId` identifies the CDF file.

Examples Open the example CDF and find out how many attributes in the file have variable scope:

```
cdfid = cdflib.open('example.cdf');

% Determine the number of attributes with variable scope
numAttrs = cdflib.getNumAttributes(cdfid)

numAttrs =

     1

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References This function corresponds to the CDF library C API routine `CDFgetNumvAttributes`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also `cdflib.getNumgAttributes`

Tutorials

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

cdflib.getNumgAttributes

Purpose

Number of attributes with global scope

Syntax

```
ngatts = cdflib.getNumgAttributes(cdfId)
```

Description

`ngatts = cdflib.getNumgAttributes(cdfId)` returns the total number of global attributes in a Common Data Format (CDF) file. `cdfId` identifies the CDF file.

Examples

Open the example CDF and find out how many global attributes are in the file:

```
cdfid = cdflib.open('example.cdf');

% Determine the number of global attributes in the file.
numgAttrs = cdflib.getNumgAttributes(cdfid)

numgAttrs =

     3

% Clean up
cdflib.close(cdfid);
```

References

This function corresponds to the CDF library C API routine `CDFgetNumgAttributes`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getNumAttributes`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose Read-only mode

Syntax

```
mode = cdflib.getReadOnlyMode(cdfId)
```

Description

`mode = cdflib.getReadOnlyMode(cdfId)` returns the read-only mode of a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

mode

Either of the following text strings or its numeric equivalent.

'READONLYon'	CDF is in read-only mode
'READONLYoff'	CDF can be modified.

To get the numeric equivalent of these text strings, use `cdflib.getConstantValue`.

Examples

Open the example CDF file and determine its current read-only status:

```
cdfId = cdflib.open('example.cdf');

mode = cdflib.getReadOnlyMode(cdfId)

mode =

READONLYoff

% Clean up.
cdflib.close(cdfId);
clear cdfId
```

cdflib.getReadOnlyMode

References

This function corresponds to the CDF library C API routine `CDFgetReadOnlyMode`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setReadOnlyMode` | `cdflib.getConstantValue`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.getStageCacheSize

Purpose

Number of cache buffers for staging

Syntax

```
numBuffers = cdflib.getStageCacheSize(cdfId)
```

Description

`numBuffers = cdflib.getStageCacheSize(cdfId)` returns the number of cache buffers used for the staging scratch file of the Common Data Format (CDF) file. For more information about cache buffers, see the *CDF User's Guide*.

`cdfId` identifies the CDF file.

Examples

Open the example CDF file and determine the number of cache buffers used for staging:

```
cdfId = cdflib.open('example.cdf');

numBuf = cdflib.getStageCacheSize(cdfId)

numBuf =

   125

% Clean up
cdflib.close(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetStageCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setStageCacheSize`

Tutorials

- "Importing Common Data File Format (CDF) Files"

- "Exporting to Common Data File Format (CDF) Files"

Purpose

Library validation mode

Syntax

```
mode = cdflib.getValidate()
```

Description

`mode = cdflib.getValidate()` returns the validation mode of the Common Data Format (CDF) library.

Output Arguments

mode

Either of the following text strings or its numeric equivalent.

'VALIDATEFILEon'	Validation mode is on. For information about validation mode, see `cdflib.setValidate`.
'VALIDATEFILEoff'	Validation mode is off.

To get the numeric equivalent of these text strings, use `cdflib.getConstantValue`.

Examples

Determine the current validation mode of the CDF library.

```
mode = cdflib.getValidate()

mode =

'VALIDATEFILEon'
```

References

This function corresponds to the CDF library C API routine `CDFgetValidate`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setValidate` | `cdflib.getConstantValue`

Tutorials

- "Importing Common Data File Format (CDF) Files"

- “Exporting to Common Data File Format (CDF) Files”

Purpose

Number of records allocated for variable

Syntax

```
numrecs = cdflib.getVarAllocRecords(cdfId, varNum)
```

Description

`numrecs = cdflib.getVarAllocRecords(cdfId, varNum)` returns the number of records allocated for a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based.

Examples

Open example CDF and get the number of records allocated for a variable:

```
cdfid = cdflib.open('example.cdf');

% Determine the number of records allocated for the
% first variable in the file.
numrecs = cdflib.getVarAllocRecords(cdfid,0)

numrecs =

    64


%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarAllocRecords`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also `cdflib.setVarAllocBlockRecords`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Blocking factor for variable

Syntax

```
blockingFactor = cdflib.getVarBlockingFactor(cdfId,varNum)
```

Description

`blockingFactor = cdflib.getVarBlockingFactor(cdfId,varNum)` returns the blocking factor for a variable in a Common Data Format (CDF) file. A variable's *blocking factor* specifies the minimum number of records the library allocates when you write to an unallocated record.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based.

Examples

Open the example CDF and determine the blocking factor of a variable.

```
cdfid = cdflib.open('example.cdf');

cdflib.getVarBlockingFactor(cdfid,0)

ans =

     0

%  Clean up
cdflib.close(cdfid)
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarBlockingFactor`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setVarBlockingFactor`

Tutorials

- "Importing Common Data File Format (CDF) Files"

- “Exporting to Common Data File Format (CDF) Files”

Purpose

Number of multifile cache buffers

Syntax

```
numBuffers = cdflib.getVarCacheSize(cdfId,varNum)
```

Description

`numBuffers = cdflib.getVarCacheSize(cdfId,varNum)` returns the number of cache buffers used for a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable identifiers are zero-based.

This function applies only to multifile format CDFs. For more information about caching, see the *CDF User's Guide*.

Examples

Create a multifile CDF and retrieve the number of buffers being used for a variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf')

% Set the format of the file to be multi-file
cdflib.setFormat(cdfid,'MULTI_FILE');

% Create a variable in the file
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Note how the library creates a separate file for the variable
ls your_file.*

your_file.cdf  your_file.z0

% Determine the number of cache buffers used with the variable
numBuf = cdflib.getVarCacheSize(cdfid,varNum)

numBuf =

     1

%  Clean up
```

```
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setVarCacheSize`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Information about compression used by variable

Syntax

```
[ctype,cparams,percent] = cdflib.getVarCompression(cdfId,
   varNum)
```

Description

`[ctype,cparams,percent] = cdflib.getVarCompression(cdfId, varNum)` returns information about the compression used for a variable in a Common Data Format (CDF) File.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

ctype

Text string identifying the type of compression. For a list of compression types, see `cdflib.setCompression`.

cparams

Any additional parameter required by the compression type.

percent

Numeric value indicating the level of compression, expressed as a percentage.

Examples

Open the example CDF file and check the compression settings of any variable.

```
cdfid = cdflib.open('example.cdf');

% Check the compression setting of any variable in the file
% The example checks the first variable (variable numbers are zero-bas
[ctype params percent] = cdflib.getVarCompression(cdfid,0)

ctype =
```

```
NO_COMPRESSION


params =

     []


percent =

   100

% Clean up
cdflib.close(cdfid);
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarCompression`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setCompression` | `cdflib.setVarCompression`

Tutorials

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

Purpose

Single value from record in variable

Syntax

```
datum = cdflib.getVarData(cdfId,varNum,recNum,indices)
```

Description

`datum = cdflib.getVarData(cdfId,varNum,recNum,indices)` returns a single value from a variable in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varNum

Numeric value identifying the variable containing the datum. Variable numbers are zero-based.

recNum

Numeric value identifying the location of the datum in the variable. In CDF terminology, this is called the *record number*. Record numbers are zero-based.

indices

Array of dimension indices within the record. Dimension indices are zero-based. If the variable has no dimensions, you can omit this parameter.

Output Arguments

datum

Value of the specified record.

Examples

Open the example CDF file and retrieve data associated with a variable:

```
cdfid = cdflib.open('example.cdf');

% Determine how many variables are in the file.
```

```
info = cdflib.inquire(cdfid);

info.numVars

ans =

  5

% Determine if the first variable has dimensions.
varinfo = cdflib.inquireVar(cdfid,0);
vardims = varinfo.dims
vardims =

  []

% Get data from variable, without specifying dimensions.
datum = cdflib.getVarData(cdfid, varnum, recnum)

datum =

  6.3146e+013

% Get dimensions of another variable in file.
varinfo = cdflib.inquireVar(cdfid,3);
vardims = varinfo.dims
vardims =

  [4 2 2]

% Retrieve the first datum in the record. Indices are zero-based.
datum = cdflib.getVarData(cdfId,3,0,[0 0 0])

info =

    30

% Clean up.
```

```
cdflib.close(cdfid);
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarData`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.putVarData` | `cdflib.getVarRecordData` | `cdflib.hyperGetVarData`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.getVarMaxAllocRecNum

Purpose

Maximum allocated record number for variable

Syntax

```
maxrec = cdflib.getVarMaxAllocRecNum(cdfId,varNum)
```

Description

`maxrec = cdflib.getVarMaxAllocRecNum(cdfId,varNum)` returns the record number of the maximum allocated record for a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers and record numbers are zero-based.

Examples

Open example CDF and get the maximum allocated record number for a variable:

```
cdfid = cdflib.open('example.cdf');

% Determine maximum record number for variable in file.
maxRecNum = cdflib.getVarMaxAllocRecNum(cdfid,0)

maxRecNum =

    63

%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarMaxAllocRecNum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarMaxWrittenRecNum`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.getVarMaxWrittenRecNum

Purpose

Maximum written record number for variable

Syntax

```
maxrec = cdflib.getVarMaxwrittenRecNum(cdfId,varNum)
```

Description

`maxrec = cdflib.getVarMaxwrittenRecNum(cdfId,varNum)` returns the record number of the maximum record written for a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers and record numbers are zero-based.

Examples

Open the example CDF, and then determine the maximum number of records written to a variable:

```
cdfid = cdflib.open('example.cdf');

% Determine the number records written to variable.
numRecs = cdflib.getVarNumRecsWritten(cdfid,0)

numRecs =

    24


% Determine the maximum record number of the records written
maxRecNum = cdflib.getVarMaxWrittenRecNum(cdfid,0)

maxRecNum =

    23

%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarMaxWrittenRecNum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarMaxAllocRecNum` | `cdflib.getVarNumRecsWritten`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.getVarsMaxWrittenRecNum

Purpose

Maximum written record number for CDF file

Syntax

```
maxrec = cdflib.getVarsMaxwrittenRecNum(cdfId)
```

Description

`maxrec = cdflib.getVarsMaxwrittenRecNum(cdfId)` returns the maximum record number written for all variables in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. Record numbers are zero-based.

Examples

Open the example CDF, and then determine the maximum number of records written to the file:

```
cdfid = cdflib.open('example.cdf');

% Determine the maximum record number of the records written
maxRecNum = cdflib.getVarsMaxWrittenRecNum(cdfid)

maxRecNum =

    0

%  Clean up
cdflib.close(cdfid)
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarsMaxWrittenRecNum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarMaxWrittenRecNum`

Tutorials

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

Purpose

Variable name, given variable number

Syntax

```
name = cdflib.getVarName(cdfId,varNum)
```

Description

`name = cdflib.getVarName(cdfId,varNum)` returns the name of the variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based.`name` is a text string specifying the name.

Examples

Open the example CDF, and then get the name of a variable in the file:

```
cdfid = cdflib.open('example.cdf');

name = cdflib.getVarName(cdfid,1)

name =

Longitude

%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarName`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.inquireVar`

Tutorials

- "Importing Common Data File Format (CDF) Files"

- "Exporting to Common Data File Format (CDF) Files"

Purpose

Variable number, given variable name

Syntax

```
varNum = cdflib.getVarNum(cdfId,varname)
```

Description

`varNum = cdflib.getVarNum(cdfId,varname)` returns the identifier (variable number) for a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. *varname* is a text string that identifies the variable. Variable names are case-sensitive.

Examples

Open example CDF, and then get the number of a variable named `Longitude`:

```
cdfid = cdflib.open('example.cdf');

varNum = cdflib.getVarNum(cdfid,'Longitude')

varNum =

     1

%  Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarNum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarName`

Tutorials

- "Importing Common Data File Format (CDF) Files"

- “Exporting to Common Data File Format (CDF) Files”

Purpose

Number of records written to variable

Syntax

```
numrecs = cdflib.getVarNumRecsWritten(cdfId,varNum)
```

Description

`numrecs = cdflib.getVarNumRecsWritten(cdfId,varNum)` returns the total number of records written to a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based.

Examples

Open the example CDF, and then determine the number of records written to a variable:

```
cdfid = cdflib.open('example.cdf');

% Determine the number of records written to the variable.
numRecs = cdflib.getVarNumRecsWritten(cdfid,0)

numRecs =

    24

%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarNumRecsWritten`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarMaxWrittenRecNum`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Pad value for variable

Syntax

```
padvalue = cdflib.getVarPadValue(cdfId,varNum)
```

Description

`padvalue = cdflib.getVarPadValue(cdfId,varNum)` returns the pad value used with a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based.

Examples

Open the example CDF, and then determine the pad value for a variable:

```
cdfid = cdflib.open('example.cdf');

% Check pad value of variable in the file.
padval = cdflib.getVarPadValue(cdfid,0)

padval =

     0

%  Clean up.
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarPadValue`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setVarPadValue`

Tutorials

- “Importing Common Data File Format (CDF) Files”

- "Exporting to Common Data File Format (CDF) Files"

Purpose

Entire record for variable

Syntax

```
data = cdflib.getVarRecordData(cdfId,varNum,recNum)
```

Description

`data = cdflib.getVarRecordData(cdfId,varNum,recNum)` returns the data in a record associated with a variable in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varNum

Numeric value that identifies the variable in the CDF file. Variable numbers are zero-based.

recNum

Numeric value that identifies the record in the variable. Record numbers are zero-based.

Output Arguments

data

Data in the record.

Examples

Open the example CDF, and then get the data associated with a record in a variable:

```
cdfid = cdflib.open('example.cdf');

% Get data in first record in first variable in file.
recData = cdflib.getVarRecordData(cdfid,0,0)

recData =

    6.3146e+013
```

```
%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarRecordData`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.putVarRecordData` | `cdflib.getVarData` | `cdflib.hyperGetVarData`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Compression reserve percentage for variable

Syntax

```
percent = cdflib.getVarReservePercent(cdfId,varNum)
```

Description

`percent = cdflib.getVarReservePercent(cdfId,varNum)` returns the compression reserve percentage for a variable in a Common Data Format (CDF) file. This operation only applies to compressed variables.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based.

Definitions

reserve percentage

Specifies how much extra space to allocate for a compressed variable. This extra space allows the variable to expand when you write additional records to the variable. If you do not specify this room for growth, the library has to move the variable to the end of the file when the size expands and the space at the original location of the variable becomes wasted space.

By default, the reserve percent is `0` (no extra space is reserved). You can specify any percentage between `1` and `100` and values greater than `100`. The value specifies the percentage of the uncompressed size of the variable.

Examples

Open the example CDF file, get the number of a compressed variable, and then determine the reserve percent for the variable.

```
cdfid = cdflib.open('example.cdf');
varnum = cdflib.getVarNum(cdfid,'Temperature');
percent = cdflib.getVarReservePercent(cdfid,varnum);
cdflib.close(cdfid);
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarReservePercent`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setVarReservePercent`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Information about how variable handles sparse records

Syntax

`stype = cdflib.getVarSparseRecords(cdfId,varNum)`

Description

`stype = cdflib.getVarSparseRecords(cdfId,varNum)` returns information about how a variable in the Common Data Format (CDF) file handles sparse records.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varNum

Numeric value that identifies the variable. Variable numbers are zero-based.

Output Arguments

stype

One of the following text strings, or its numeric equivalent, that specifies how the variable handles sparse records.

Text String	Description
'NO_SPARSERECORDS'	No sparse records.
'PAD_SPARSERECORDS'	For sparse records, the library uses the variable's pad value when reading values from a missing record.
'PREV_SPARSERECORDS'	For sparse records, the library uses values from the previous existing record when reading values from a missing record. If there is no previous existing record, the library uses the variable's pad value.

To get the numeric equivalent of these text strings, use `cdflib.getConstantValue`.

Examples

Open the example CDF, and then get the sparse record type of a variable in the file:

```
cdfid = cdflib.open('example.cdf');

stype = cdflib.getVarSparseRecords(cdfid,0)

stype =

NO_SPARSERECORDS

%Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFgetzVarSparseRecords`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setVarSparseRecords`

Tutorials

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

Purpose

Common Data Format (CDF) library version and release information

Syntax

```
[version,release,increment] = cdflib.getVersion(cdfId)
```

Description

`[version,release,increment] = cdflib.getVersion(cdfId)` returns information about the version of the Common Data Format (CDF) library used to create a CDF file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

version

Numeric value indicating the version number of the CDF library.

release

Numeric value indicating the release number of the CDF library.

increment

Numeric value indicating the increment number of the CDF library.

Examples

Open the example CDF file, and then find out the version of the CDF library used to create it:

```
cdfId = cdflib.open('example.cdf');

[version, release, increment] = cdflib.getVersion(cdfId)

version =

     2

release =
```

```
     7

increment =

     8

% Clean up
cdflib.close(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFgetVersion`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getLibraryVersion`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Read hyperslab of data from variable

Syntax

```
data = cdflib.hyperGetVarData(cdfId,varNum,recSpec,dimSpec)
```

Description

`data = cdflib.hyperGetVarData(cdfId,varNum,recSpec,dimSpec)` reads a hyperslab of data from a variable in the Common Data Format (CDF) file. Hyper access allows more than one value to be read from or written to a variable with a single call to the CDF library.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varNum

Number identifying the variable containing the datum.

recSpec

Three-element array, `[RSTART RCOUNT RSTRIDE]`, where `RSTART`, `RCOUNT`, and `RSTRIDE` are scalar values specifying the starting record, number of records to read, and the sampling interval or stride between records. Record numbers are zero-based.

dimSpec

Three-element cell array, `{DSTART DCOUNT DSTRIDE}`, where `DSTART`, `DCOUNT`, and `DSTRIDE` are *n*-element vectors that describe the start, number of values along each dimension, and sampling interval along each dimension. If the hyperslab has zero dimensions, you can omit this parameter. Dimension indices are zero-based.

Examples

Open the example CDF file, and then get all the data associated with a variable:

```
cdfid = cdflib.open('example.cdf');
```

```
% Determine the number of records allocated for the first variable in the
maxRecNum = cdflib.getVarMaxWrittenRecNum(cdfid,0);

% Retrieve all data in records for variable.
data = cdflib.hyperGetVarData(cdfid,0,[0 maxRecNum 1]);

%  Clean up
cdflib.close(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFhyperGetzVarData`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.hyperPutVarData`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Write hyperslab of data to variable

Syntax

`cdflib.hyperPutVarData(cdfId,varNum,recSpec,dimSpec,data)`

Description

`cdflib.hyperPutVarData(cdfId,varNum,recSpec,dimSpec,data)` writes a hyperslab of data to a variable in a Common Data Format (CDF) file. Hyper access allows more than one value to be read from or written to a variable with a single call to the CDF library.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varNum

Specifies the variable containing the datum.

recSpec

Three-element array described by `[RSTART RCOUNT RSTRIDE]`, where `RSTART`, `RCOUNT`, and `RSTRIDE` are scalar values giving the start, number of records, and sampling interval (or stride) between records. Record indices are zero-based.

dimSpec

Three-element cell array described by `{DSTART DCOUNT DSTRIDE}`, where `DSTART`, `DCOUNT`, and `DSTRIDE` are n-element vectors that describe the start, number of values along each dimension, and sampling interval along each dimension. If the hyperslab has zero dimensions, you can omit this parameter. Dimension indices are zero-based.

data

Data to write to the variable.

Examples

Create a CDF, create a variable, and then write a slab of data to the variable. To run this example, you must be in a writable folder.

cdflib.hyperPutVarData

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Grades','cdf_int1',1,[],true,[]);

% Write data to the variable
cdflib.hyperPutVarData(cdfid,varNum,0,[],int8(98))


%Clean up
cdflib.delete(cdfid);
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFhyperzPutVarData`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.hyperGetVarData`

Tutorials

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

Purpose

Basic characteristics of Common Data Format (CDF) file

Syntax

```
info = cdflib.inquire(cdfId)
```

Description

`info = cdflib.inquire(cdfId)` returns basic information about a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

Output Arguments

info

A structure containing the following fields:

Field	Description
`encoding`	Encoding of the variable data and attribute entry data
`majority`	Majority of the variable data
`maxRec`	Maximum record number written to a CDF variable
`numVars`	Number of CDF variables
`numvAttrs`	Number of attributes with variable scope
`numgAttrs`	Number of attributes with global scope

Examples

Open the example CDF file, and then get basic information about the file:

```
cdfId = cdflib.open('example.cdf');

info = cdflib.inquire(cdfId)

info =
```

```
      encoding: 'IBMPC_ENCODING'
      majority: 'ROW_MAJOR'
        maxRec: 23
       numVars: 5
     numvAttrs: 1
     numgAttrs: 3
```

References

This function corresponds to the CDF library C API routines `CDFinquireCDF` and `CDFgetNumgAttributes`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.inquireVar`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Information about attribute

Syntax

```
info = cdflib.inquireAttr(cdfId,attrNum)
```

Description

`info = cdflib.inquireAttr(cdfId,attrNum)` returns information about an attribute in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

attrNum

Numeric value that identifies the attribute in the file. Attribute numbers are zero-based.

Output Arguments

info

Structure containing the following fields.

Field	Description
`name`	Attribute's name
`scope`	Either `'GLOBAL_SCOPE'` or `'VARIABLE_SCOPE'`
`maxgEntry`	The maximum entry number used for global attributes.
`maxEntry`	The maximum entry number used for attributes with variable scope.

Examples

Open the example CDF, and then get information about the first attribute in the file.

```
cdfid = cdflib.open('example.cdf');
```

```
% Get information about an attribute
info = cdflib.inquireAttr(cdfid,0)

info =

         name: 'SampleAttribute'
        scope: 'GLOBAL_SCOPE'
    maxgEntry: 4
     maxEntry: -1

% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFinquireAttr`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.inquireAttrgEntry` | `cdflib.inquireAttrEntry`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Information about entry in attribute with variable scope

Syntax

```
[datatype,numElements] = cdflib.inquireAttrEntry(cdfId,
   attrNum,entryNum)
```

Description

`[datatype,numElements] = cdflib.inquireAttrEntry(cdfId,attrNum,entryNum)` returns the data type and the number of elements for an attribute entry in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

attrNum

Numeric value identifying an attribute in the file. Attribute numbers are zero-based. The attribute must have variable scope.

entryNum

Numeric value identifying the entry in the attribute. Entry number are zero-based.

Output Arguments

datatype

Text string identifying a CDF data type. For a list of CDF data types, see `cdflib.putAttrEntry`

numElements

Numeric value indicating the number of elements in the entry.

Examples

Open example CDF, and then get information about entries associated with an attribute in the file:

```
cdfid = cdflib.open('example.cdf');
```

```
% The fourth attribute is of variable scope.
attrscope = cdflib.getAttrScope(cdfid,3)

attrscope =

VARIABLE_SCOPE

% Get information about the first entry for this attribute
[dtype numel] = cdflib.inquireAttrEntry(cdfid,3,0)

dtype =

cdf_char


numel =

    10


% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFinquireAttrzEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.inquireAttr` | `cdflib.getAttrScope`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Information about entry in attribute with global scope

Syntax

```
[datatype,numElements] = cdflib.inquireAttrgEntry(cdfId,
   attrNum,entryNum)
```

Description

`[datatype,numElements] = cdflib.inquireAttrgEntry(cdfId,attrNum,entryNum)` returns the data type and the number of elements for a global attribute entry in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

attrNum

Numeric value identifying an attribute in the file. Attribute numbers are zero-based. The attribute must have global scope.

entryNum

Numeric value identifying the entry in the attribute. Entry number are zero-based.

Output Arguments

datatype

Text string identifying a CDF data type. For a list of CDF data types, see `cdflib.putAttrgEntry`

numElements

Numeric value indicating the number of elements in the entry.

Examples

Open the example CDF, and then get information about entries associated with a global attribute in the file.

```
cdfid = cdflib.open('example.cdf');
```

```
% Any of the first three attributes have global scope.
attrscope = cdflib.getAttrScope(cdfid,0)

attrscope =

GLOBAL_SCOPE

% Get information about the first entry for this attribute
[dtype numel] = cdflib.inquireAttrgEntry(cdfid,0,0)

dtype =

cdf_char


numel =

    23


% Clean up
cdflib.close(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFinquireAttrgEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.inquireAttr` | `cdflib.inquireAttrEntry`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Information about variable

Syntax

```
info = cdflib.inquireVar(cdfId,varNum)
```

Description

`info = cdflib.inquireVar(cdfId,varNum)` returns information about a variable in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varNum

Numeric value that identifies the variable. Variable numbers are zero-based.

Output Arguments

info

Structure containing the following fields.

Field	Description
`name`	Name of the variable
`datatype`	Data type
`numElements`	Number of elements of the datatype
`dims`	Sizes of the dimensions
`recVariance`	Record variance
`dimVariance`	Dimension variances

Record and dimension variances affect how the library physically stores variable data. For example, if a variable has a record variance of `VARY`, the library physically stores each record. If the record variance is `NOVARY`, the library only stores one record.

Examples

Open the example CDF file and get information about a variable.

```
cdfid = cdflib.open('example.cdf');

% Determine if the file contains variables
info  = cdflib.inquireVar(cdfid,1)

info =

           name: 'Longitude'
       datatype: 'cdf_int1'
    numElements: 1
           dims: [2 2]
    recVariance: 0
    dimVariance: [1 0]
```

References

This function corresponds to the CDF library C API routine `CDFinquirezVar`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.inquire`

Tutorials

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

Purpose

Open existing Common Data Format (CDF) file

Syntax

`cdfId = cdflib.open(filename)`

Description

`cdfId = cdflib.open(filename)` opens an existing Common Data Format (CDF) file. *`filename`* is a text string that identifies the file.

This function returns a CDF file identifier, `cdfId`.

All CDF files opened this way have the `zMode` set to `zModeon2`. Refer to the *CDF User's Guide* for information about `zModes`.

Examples

Open the example CDF file:

```
cdfId = cdflib.open('example.cdf');

% Clean up
cdflib.close(cdfId)

clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFopenCDF`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.close` | `cdflib.create`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.putAttrEntry

Purpose

Write value to entry in attribute with variable scope

Syntax

```
cdflib.putAttrEntry(cdfId,attrNum,entryNum,CDFDataType,
   entryVal)
```

Description

`cdflib.putAttrEntry(cdfId,attrNum,entryNum,CDFDataType,entryVal)` writes a value to an attribute entry in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

attrNum

Number identifying attribute. The attribute must have variable scope. Attribute numbers are zero-based.

entryNum

Number identifying entry. Entry numbers are zero-based.

CDFdatatype

One of the following text strings, or its numeric equivalent, that specify the data type of the attribute entry.

CDF Data Type	MATLAB Equivalent
`CDF_BYTE`	1-byte, signed integer
`CDF_CHAR`	1 byte, signed character data type that maps to the MATLAB `char` class
`CDF_INT1`	1-byte, signed integer.
`CDF_UCHAR`	1 byte, unsigned character data type that maps to the MATLAB `uint8` class
`CDF_UINT1`	1-byte, unsigned integer

CDF Data Type	MATLAB Equivalent
CDF_INT2	2-byte, signed integer
CDF_UINT2	2-byte, unsigned integer.
CDF_INT4	4-byte, signed integer
CDF_UINT4	4-byte, unsigned integer
CDF_FLOAT	4-byte, floating point
CDF_REAL4	4-byte, floating point
CDF_REAL8	8-byte, floating point.
CDF_DOUBLE	8-byte, floating point
CDF_EPOCH	8-byte, floating point
CDF_EPOCH16	two 8-byte, floating point

entryVal

Data to be written to attribute entry.

Examples

Create a CDF and create an attribute with variable scope in the file. Write a value to an entry in the attribute. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Initially the file contains no attributes, global or variable.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 0
    numvAttrs: 0
```

```
    numgAttrs: 0

% Create an attribute of variable scope in the file.
attrNum = cdflib.createAttr(cdfid,'Another Attribute','variable_scope');

% Write a value to an entry for the attribute
cdflib.putAttrEntry(cdfid,attrNum,0,'CDF_CHAR','My Variable Attribute Tes

% Get the value of the global attribute entry
value = cdflib.getAttrEntry(cdfid,attrNum,0)

value =

My Variable Attribute Test


% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFputAttrzEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrEntry` | `cdflib.putAttrgEntry` | `cdflib.getAttrgEntry` | `cdflib.getConstantValue`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Write value to entry in attribute with global scope

Syntax

```
cdflib.putAttrgEntry(cdfId,attrNum,entryNum,cdfDataType,
   entryVal)
```

Description

`cdflib.putAttrgEntry(cdfId,attrNum,entryNum,cdfDataType,entryVal)` writes a value to a global attribute entry in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

attrNum

Number identifying attribute. Attribute numbers are zero-based. The attribute must have global scope.

entryNum

Number identifying entry. Entry numbers are zero-based.

CDFdatatype

One of the following text strings that specify the data type of the attribute entry, or its numeric equivalent.

CDF Data Type	MATLAB Equivalent
`CDF_BYTE`	1-byte, signed integer
`CDF_CHAR`	1 byte, signed character data type that maps to the MATLAB `char` class
`CDF_INT1`	1-byte, signed integer.
`CDF_UCHAR`	1 byte, unsigned character data type that maps to the MATLAB `uint8` class
`CDF_UINT1`	1-byte, unsigned integer

CDF Data Type	MATLAB Equivalent
CDF_INT2	2-byte, signed integer
CDF_UINT2	2-byte, unsigned integer.
CDF_INT4	4-byte, signed integer
CDF_UINT4	4-byte, unsigned integer
CDF_FLOAT	4-byte, floating point
CDF_REAL4	4-byte, floating point
CDF_REAL8	8-byte, floating point.
CDF_DOUBLE	8-byte, floating point
CDF_EPOCH	8-byte, floating point
CDF_EPOCH16	two 8-byte, floating point

entryVal

Data to be written to global attribute entry.

Examples

Create a CDF and create a global attribute in the file. Write a value to an entry in the attribute. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Initially the file contains no attributes, global or variable.
info = cdflib.inquire(cdfid)

info =

     encoding: 'IBMPC_ENCODING'
     majority: 'ROW_MAJOR'
       maxRec: -1
      numVars: 0
    numvAttrs: 0
```

```
    numgAttrs: 0

% Create a global attribute in the file.
attrNum = cdflib.createAttr(cdfid,'Purpose','global_scope');

% Write a value to an entry for the global attribute
cdflib.putAttrgEntry(cdfid,attrNum,0,'CDF_CHAR','My Test');

% Get the value of the global attribute entry
value = cdflib.getAttrgEntry(cdfid,attrNum,0)

value =

My Test

% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFputAttrgEntry`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getAttrgEntry` | `cdflib.putAttrEntry` | `cdflib.getAttrEntry` | `cdflib.getConstantValue`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.putVarData

Purpose

Write single value to variable

Syntax

```
cdflib.putVarData(cdfId,varNum,recNum,indices,datum)
```

Description

`cdflib.putVarData(cdfId,varNum,recNum,indices,datum)` writes a single value to a variable in a Common Data File (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varNum

Numeric value that identifies the variable to which you want to write the datum. Variable numbers are zero-based.

recNum

Numeric value that identifies the record to which you want to write the datum. Record numbers are zero-based.

dims

Dimension indices within the record. Dimension indices are zero-based.

datum

Data to be written to the variable.

Examples

Create a CDF, create a variable in the CDF and write data to the variable. To run this example, you must have write permission in the current folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Grades','cdf_int1',1,[],true,[]);
```

```
% Write some data to the variable
cdflib.putVarData(cdfid,varNum,0,[],int8(98))

% Read the value from the variable.
datum = cdflib.getVarData(cdfid,varNum,0)

datum =

   98

%Clean up
cdflib.delete(cdfid);
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFputzVarData`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarData` | `cdflib.getVarRecordData` | `cdflib.hyperGetVarData`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.putVarRecordData

Purpose

Write entire record to variable

Syntax

```
cdflib.putVarRecordData(cdfId,varNum,recNum,recordData)
```

Description

`cdflib.putVarRecordData(cdfId,varNum,recNum,recordData)` writes data to a record in a variable in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varNum

Numeric value that identifies the variable to which you want to write the datum. Variable numbers are zero-based.

recNum

Numeric value identifying the location of the datum in the variable. Record numbers are zero-based.

recordData

Data to be written to the variable.

Examples

Create a CDF, create a variable, and write an entire record of data to the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Grades','cdf_int1',1,[],true,[]);

% Write some data to the variable
cdflib.putVarRecordData(cdfid,varNum,0,int8(98))
```

```
% Read the value from the variable.
datum = cdflib.getVarData(cdfid,varNum,0)

datum =

   98

%Clean up
cdflib.delete(cdfid);
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFputzVarRecordData`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarRecordData` | `cdflib.putVarData` | `cdflib.hyperPutVarData`

Tutorials

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

cdflib.renameAttr

Purpose Rename existing attribute

Syntax

```
cdflib.renameAttr(cdfId,attrNum,newName)
```

Description `cdflib.renameAttr(cdfId,attrNum,newName)` renames an attribute in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `attrNum` is a numeric value that identifies the attribute. Attribute numbers are zero-based. `newName` is a text string that specifies the name you want to assign to the attribute.

Examples Create a CDF, create an attribute in the CDF, and then rename the attribute. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create an attribute
attrNum = cdflib.createAttr(cdfid,'Purpose','global_scope');

% Rename the attribute
cdflib.renameAttr(cdfid, attrNum, 'NewPurpose');

% Check the name of the attribute
attrName = cdflib.getAttrName(cdfid,anum)

attrName =

NewPurpose

% Clean up
cdflib.delete(cdfid);

clear cdfid
```

References This function corresponds to the CDF library C API routine `CDFrenameAttr`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createAttr`

Tutorials

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

cdflib.renameVar

Purpose

Rename existing variable

Syntax

```
cdflib.renameVar(cdfId,varNum,newName)
```

Description

`cdflib.renameVar(cdfId,varNum,newName)` renames a variable in a Common Data Format (CDF) file.

`cdfId` identifies the CDF file. `varNum` is a numeric value that identifies the variable. Variable numbers are zero-based. `newName` is a text string that specifies the name you want to assign to the variable.

Examples

Create a CDF, create a variable in the CDF, and then rename the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Get the name of the variable.
name = cdflib.getVarName(cdfid,varNum)

name =

Time

% Rename the variable
cdflib.renameVar(cdfid,varNum,'NewName');

% Check the new name.
name = cdflib.getVarName(cdfid,varNum)

name =

NewName

%  Clean up
```

```
cdflib.delete(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFrenamezVar`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createVar`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.setCacheSize

Purpose

Specify number of dotCDF cache buffers

Syntax

`cdflib.setCacheSize(cdfId,numBuffers)`

Description

`cdflib.setCacheSize(cdfId,numBuffers)` specifies the number of cache buffers the CDF library uses for an open dotCDF file. A *dotCDF* file is a file with the `.cdf` file extension.

`cdfId` identifies an open CDF file. `numBuffers` is a numeric value that specifies the number of buffers.

For information about cache schemes, see the *CDF User's Guide*.

Examples

Create a CDF file and set the cache size. To run this example, you must have write permission in your current folder.

```
cdfId = cdflib.create('your_file.cdf');

% Get the default cache size
numBuf = cdflib.getCacheSize(cdfid)

numBuf =

   300

% Specify a cache size
cdflib.setCacheSize(cdfid,150)

% Check the cache size again
numBuf =  cdflib.getCacheSize(cdfid)

numBuf =

   150

% Clean up
cdflib.delete(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getCacheSize`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.setChecksum

Purpose

Specify checksum mode

Syntax

`cdflib.setChecksum(cdfId,mode)`

Description

`cdflib.setChecksum(cdfId,mode)` specifies the checksum mode of a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

mode

Either of the following text strings, or its numeric equivalent. To get the numeric equivalent of these values, use `cdflib.getConstantValue`.

'MD5_CHECKSUM'	Sets file checksum to MD5 checksum.
'NO_CHECKSUM'	File does not use a checksum.

Examples

Create a CDF file and set the checksum mode. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('mycdf.cdf');

% Check initial value of checksum.
mode = cdflib.getChecksum(cdfid)

NO_CHECKSUM

cdflib.setChecksum(cdfid,'MD5_CHECKSUM')

% Verify the setting
mode = cdflib.getChecksum(cdfid)

MD5_CHECKSUM
```

References

This function corresponds to the CDF library C API routine `CDFsetChecksum`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getChecksum` | `cdflib.getConstantValue`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.setCompression

Purpose

Specify compression settings

Syntax

`cdflib.setCompression(cdfId,ctype,cparms)`

Description

`cdflib.setCompression(cdfId,ctype,cparms)` specifies compression settings of a Common Data Format (CDF) file.

This function sets the compression for the CDF file itself, not that of any variables in the file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

ctype

One of the following text strings, or its numeric equivalent, specifying compression type.

Text String	Compression Type
`'NO_COMPRESSION'`	No compression
`'RLE_COMPRESSION'`	Run-length encoding compression
`'HUFF_COMPRESSION'`	Huffman compression
`'AHUFF_COMPRESSION'`	Adaptive Huffman compression
`'GZIP_COMPRESSION'`	GNU's zip compression

To get the numeric equivalent, use `cdflib.getConstantValue`.

cparms

Optional parameter specifying any additional parameters required by the compression type. Currently, the only compression type that uses this parameter is `'GZIP_COMPRESSION'`. For this compression type, use `cparms` to specify the level of compression as a numeric value between 1 and 9.

Examples

Create a CDF file and set the compression setting of the file. To run this example, your current folder must be writable.

```
cdfId = cdflib.create('your_file.cdf');

% Determine the file's default compression setting
[ctype, cparms, cpercent ] = cdflib.getCompression(cdfId)

ctype =

NO_COMPRESSION

cparms =

     []

cpercent =

   100

% Specify new compression setting
cdflib.setCompression(cdfId,'HUFF_COMPRESSION');

% Check the file's compression setting.
[ctype, cparms, cpercent ] = cdflib.getCompression(cdfId)

ctype =

HUFF_COMPRESSION

cparms =

OPTIMAL_ENCODING_TREES


cpercent =
```

```
     0

% Clean up
cdflib.delete(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetCompression`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getCompression` | `cdflib.getConstantValue`

Tutorials

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

Purpose

Specify number of compression cache buffers

Syntax

`cdflib.setCompressionCacheSize(cdfId,numBuffers)`

Description

`cdflib.setCompressionCacheSize(cdfId,numBuffers)` specifies the number of cache buffers used for the compression scratch CDF file. For more information about CDF cache schemes, see the *CDF User's Guide*.

`cdfId` identifies the CDF file. `numBuffers` specifies the number of buffers.

Examples

Create a CDF file and specify the number of compression cache buffers used. To run this example you must be in a writable folder.

```
cdfId = cdflib.create('your_file.cdf');

% Get the current number of compression cache buffers
numBuf = cdflib.getCompressionCacheSize(cdfId)

numBuf =

    80

% Set a new value
cdflib.setCompressionCacheSize(cdfId,100)

% Check the new value
numBuf = cdflib.getCompressionCacheSize(cdfId)

numBuf =

   100

% Clean up
cdflib.delete(cdfId)
clear cdfId
```

cdflib.setCompressionCacheSize

References

This function corresponds to the CDF library C API routine `CDFsetCompressionCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getCompressionCacheSize`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Set backward compatibility mode

Syntax

```
cdflib.getFileBackward(mode)
```

Description

`cdflib.getFileBackward(mode)` sets the backward compatibility mode to the value specified by `mode`.

Tips

- Setting backward compatibility mode affects only your current MATLAB session, or until you call `cdflib.setFileBackward` again.

Input Arguments

mode

One of the following text strings:

BACKWARDFILEon	Set backward compatibility mode on.
BACKWARDFILEoff	Set backward compatibility mode off.

Default: `BACKWARDFILEoff`

Definitions

backward compatibility mode

When specified, ensures that any new CDF file created using CDF V3.0 (or later) will be readable by clients using version 2.7 of the CDF library. CDF 3.0 and later releases use a 64-bit file offset to allow for files greater than 2G bytes in size. CDF library versions released before CDF 3.0 use a 32-bit file offset.

Examples

Set backward compatibility mode and then check the value.

```
cdflib.setFileBackward('BACKWARDFILEon');

mode = cdflib.getFileBackward

mode =

BACKWARDFILEon
```

References

This function corresponds to the CDF library C API routine `CDFsetFileBackward`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getFileBackward` | `cdflib.getConstantValue`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"
-

Purpose

Specify format of Common Data Format (CDF) file

Syntax

`cdflib.setFormat(cdfId,format)`

Description

`cdflib.setFormat(cdfId,format)` specifies the format of a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

format

Either of the following text strings, or its numeric equivalent.

'SINGLE_FILE'	The CDF consists of only one file. This is the default file format
'MULTI_FILE'	The CDF consists of one header file for control and attribute data and one additional file for each variable in the CDF.

To get the numeric equivalent of these values, use `cdflib.getConstantValue`.

Examples

Create a CDF file and specify its format. To run this example, you must have write permission in your current folder.

```
cdfId = cdflib.create('mycdffile.cdf');

% Specify multifile format.
cdflib.setFormat(cdfId, 'MULTI_FILE');

% Check format.
 format = cdflib.getFormat(cdfId)

format =
```

```
MULTI_FILE

% Clean up
cdflib.delete(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetFormat`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getFormat` | `cdflib.getConstantValue`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Specify majority of variables

Syntax

`cdflib.setMajority(cdfId,majority)`

Description

`cdflib.setMajority(cdfId,majority)` specifies the majority of variables in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

majority

Either of the following text strings, or its numeric equivalent.

'ROW_MAJOR'	C-like array ordering for variable storage. The first dimension in each variable array varies the slowest. This is the default.
'COLUMN_MAJOR'	Fortran-like array ordering for variable storage. The first dimension in each variable array varies the fastest.

To get the numeric equivalent of these values, use `cdflib.getConstantValue`.

Examples

Create a CDF file and specify the majority used by variables in the file. To run this example, you must have write permission in your current folder.

```
cdfId = cdflib.create('your_file.cdf')

% Specify the majority used by variables in the file
cdflib.setMajority(cdfId,'COLUMN_MAJOR');

% Check the majority value
majority = cdflib.getMajority(cdfId)
```

```
majority =

COLUMN_MAJOR

% Clean up
cdflib.delete(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetMajority`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getMajority`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Specify read-only mode

Syntax

`cdflib.setReadOnlyMode(cdfId,mode)`

Description

`cdflib.setReadOnlyMode(cdfId,mode)` specifies the read-only mode of a Common Data Format (CDF) file.

After you open a CDF file, you can put the file into read-only mode to prevent accidental modification.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

mode

Either of the following text strings or its numeric equivalent.

'READONLYon'	CDF file is read-only
'READONLYoff'	CDF file is modifiable.

To get the numeric equivalent of these mode values, use `cdflib.getConstantValue`.

Examples

Open the example CDF file and set the file to read-only mode.

```
cdfId = cdflib.open('example.cdf');

% Set the  file to READONLY mode
cdflib.setReadOnlyMode(cdfId,'READONLYon')

% Check read-only status of file again.
mode = cdflib.getReadOnlyMode(cdfId)

mode =

READONLYon
```

```
% Clean up
cdflib.close(cdfId)
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetReadOnlyMode`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getReadOnlyMode` | `cdflib.getConstantValue`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose Specify number of staging cache buffers for Common Data Format (CDF) file

Syntax

```
cdflib.setStageCacheSize(cdfId,numBuffers)
```

Description `cdflib.setStageCacheSize(cdfId,numBuffers)` specifies the number of staging cache buffers for a Common Data Format (CDF) file. For information about CDF cache schemes, see the *CDF User's Guide*.

`cdfId` identifies the CDF file. `numBuffers` is a numeric value that specifies the number of buffers.

Examples Open the example CDF file and specify the number of cache buffers used.

```
cdfId = cdflib.open('example.cdf');

% Get current number of staging cache buffers
size = cdflib.getStageCacheSize(cdfId)

size =

   125

% Specify new cache size value.
cdflib.setStageCacheSize(cdfId, 200)

% Get size again.
size = cdflib.getStageCacheSize(cdfId)

size =

   200

% Clean up
cdflib.close(cdfId)
```

```
clear cdfId
```

References

This function corresponds to the CDF library C API routine `CDFsetStageCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getStageCacheSize`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Specify library validation mode

Syntax

`cdflib.setValidate(mode)`

Description

`cdflib.setValidate(mode)` specifies the validation mode of the Common Data Format (CDF) library. Specify the validation mode before opening any files.

Input Arguments

mode

Either of the following text strings, or its numeric equivalent:

'VALIDATEFILEon'	Turns validation mode on. With validation mode on, the library performs sanity checks on the data fields in the CDF' file's internal data structures to make sure that the values are within valid ranges and consistent with the defined values/types/entries. This mode also ensures that variable and attribute associations within the file are valid. Note, however, that enabling this mode will, in most cases, slow down the file opening process, especially for large or very fragmented files.
'VALIDATEFILEoff'	Turns validation mode off.

To get the numeric equivalent of these values, use `cdflib.getConstantValue`.

Examples

Set the validation mode of the CDF library.

```
cdflib.setValidate('VALIDATEFILEon');
```

References

This function corresponds to the CDF library C API routine `CDFsetValidate`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getValidate` | `cdflib.getConstantValue`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Specify range of records to be allocated for variable

Syntax

```
cdflib.setVarAllocBlockRecords(cdfId,varNum,firstrec,lastrec)
```

Description

`cdflib.setVarAllocBlockRecords(cdfId,varNum,firstrec,lastrec)` specifies a range of records you want to allocate (but not write) for a variable in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varNum

Numeric value identifying a variable in the file. Variable identifiers (variable numbers) are zero-based.

firstRec

Numeric value identifying the record at which to start allocating. Record numbers are zero-based.

lastRec

Numeric value identifying the record at which to stop allocating. Record numbers are zero-based.

Examples

Create a CDF, create a variable in the CDF, and then specify the number of records to allocate for the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Grades','cdf_int1',1,[],true,[]);

% Specify the number of records to allocate.
```

```
cdflib.setVarAllocBlockRecords(cdfid,varNum,1,10);


%  Clean up
cdflib.delete(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarAllocBlockRecords`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarAllocRecords`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Specify blocking factor for variable

Syntax

```
cdflib.setVarBlockingFactor(cdfId,varNum,blockingFactor)
```

Description

`cdflib.setVarBlockingFactor(cdfId,varNum,blockingFactor)` specifies the blocking factor for a variable in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varNum

Numeric value identifying a variable in the file. Variable numbers are zero-based.

blockingFactor

Numeric value that specifies the number of records to allocate when writing to an unallocated record.

Definitions

blocking factor

A variable's *blocking factor* specifies the minimum number of records the library allocates when you write to an unallocated record. If you specify a fractional blocking factor, the library rounds the value down.

Examples

Create a CDF, create a variable in the CDF, and then set the blocking factor used with the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);
```

```
% Get the current blocking factor used with the variable
bFactor = cdflib.getVarBlockingFactor(cdfid,varNum)

bFactor =

    0

% Change the blocking factor for the variable
cdflib.setVarBlockingFactor(cdfid,varNum,10);

% Check the new blocking factor .
bFactor = cdflib.getVarBlockingFactor(cdfid,varNum)

bFactor =

    10

%  Clean up
cdflib.delete(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarBlockingFactor`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarBlockingFactor`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Specify number of multi-file cache buffers for variable

Syntax

`cdflib.setVarCacheSize(cdfId,varNum,numBuffers)`

Description

`cdflib.setVarCacheSize(cdfId,varNum,numBuffers)` specifies the number of cache buffers the CDF library uses for a variable in a Common Data Format (CDF) file.

This function is only used with multifile format CDF files. It does not apply to single-file format CDFs. For more information about caching, see the *CDF User's Guide*.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varNum

Numeric value identifying a variable in the file. Variable identifiers (variable numbers) are zero-based.

numBuffers

Numeric value identifying the number of cache buffers to use.

Examples

Create a multifile CDF, and then retrieve the number of buffers being used for a variable:

```
cdfid = cdflib.create('your_file.cdf')

% Set the format of the file to be multi-file
cdflib.setFormat(cdfid,'MULTI_FILE');

% Create a variable in the file
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Note how the library creates a separate file for the variable
```

```
ls your_file.*

your_file.cdf  your_file.z0

% Determine the number of cache buffers used with the variable
numBuf = cdflib.getVarCacheSize(cdfid,varNum)

numBuf =

     1

% Increase the number of cache buffers used.
cdflib.setVarCacheSize(cdfid,varNum,5)

% Check the number of cache buffers used with the variable.
numBuf = cdflib.getVarCacheSize(cdfid,varNum)

numBuf =

     5

%  Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarCacheSize` | `cdflib.setVarsCacheSize`

Tutorials

- “Importing Common Data File Format (CDF) Files”

- "Exporting to Common Data File Format (CDF) Files"

cdflib.setVarCompression

Purpose

Specify compression settings used with variable

Syntax

```
cdflib.setVarCompression(cdfId,varNum,ctype,cparams)
```

Description

`cdflib.setVarCompression(cdfId,varNum,ctype,cparams)` configures the compression setting for a variable in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varNum

Numeric value identifying a variable in the file. Variable identifiers (variable numbers) are zero-based.

ctype

One of the following text strings, or its numeric equivalent, specifying the compression type.

Text String	Compression Type
'NO_COMPRESSION'	No compression.
'RLE_COMPRESSION'	Run-length encoding compression
'HUFF_COMPRESSION'	Huffman compression
'AHUFF_COMPRESSION'	Adaptive Huffman compression
'GZIP_COMPRESSION'	GNU's zip compression

cparams

Optional parameter specifying any additional parameters required by the compression type. Currently, the only compression type that uses this parameter is `'GZIP_COMPRESSION'`. For this compression type,

you use `cparms` to specify the level of compression as a numeric value between 1 and 9.

Examples

Create a CDF, create a variable, and then set the compression used by the variable. To run this example, you must be in a folder with execute permission.

```
cdfid = cdflib.create('mycdf.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Specify the compression used by the variable.
cdflib.setVarCompression(cdfid,0,'GZIP_COMPRESSION',8)

% Check the compression setting of the variable
[ctype params percent] = cdflib.getVarCompression(cdfid,0)

ctype =

GZIP_COMPRESSION

params =

     8

percent =

     0

% Clean up
cdflib.delete(cdfid);
clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarCompression`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.setCompression` | `cdflib.getVarCompression`

Tutorials

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

Purpose

Specify initial number of records written to variable

Syntax

`cdflib.setVarInitialRecs(cdfId,varNum,numrecs)`

Description

`cdflib.setVarInitialRecs(cdfId,varNum,numrecs)` specifies the initial number of records to write to a variable in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varNum

Numeric value identifying a variable in the file. Variable numbers are zero-based.

numRecs

Numeric value specifying the number of records to write.

Examples

Create a CDF, create a variable, and then specify the number of records to write for the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Grades','cdf_int1',1,[],true,[]);

% Specify the number of records to write for the variable
cdflib.setVarInitialRecs(cdfid,varNum,100);

recsWritten = cdflib.getVarNumRecsWritten(cdfid,varNum)

recsWritten =
```

```
   100

%  Clean up
cdflib.delete(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarInitialRecs`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.createVar`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Specify pad value used with variable

Syntax

```
cdflib.setVarPadValue(cdfId,varNum,padvalue)
```

Description

`cdflib.setVarPadValue(cdfId,varNum,padvalue)` specifies the pad value used with a variable in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varNum

Numeric value identifying a variable in the file. Variable numbers are zero-based.

padValue

Value to use a pad value for the variable. The data type of the pad value must match the data type of the variable.

Examples

Create a CDF, create a variable in the CDF, and then set the pad value used with the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Get the current pad value used with the variable
padval = cdflib.getVarPadValue(cdfid,varNum)

padval =

    0
```

```
% Change the pad value for the variable
cdflib.setVarPadValue(cdfid,varNum,int8(1));

% Check the new pad value.
padval = cdflib.getVarPadValue(cdfid,varNum)

padval =

    1

%  Clean up
cdflib.delete(cdfid)

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarPadValue`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarPadValue`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

cdflib.SetVarReservePercent

Purpose

Specify reserve percentage for variable

Syntax

```
cdflib.setVarReservePercent(cdfId,varNum,percent)
```

Description

`cdflib.setVarReservePercent(cdfId,varNum,percent)` specifies the compression reserve percentage for a variable in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varNum

Numeric value identifying a variable in the file. Variable identifiers (variable numbers) are zero-based.

percent

Numeric value specifying the amount of extra space to allocate for a compressed variable, experssed as a percentage. You can specify values between`0` (no extra space is reserved) and `100`, or values greater than `100`. The value specifies the percentage of the uncompressed size of the variable. If you specify a fractional reserve percentages, the library rounds the value down.

Definitions

reserve percentage

Specifies how much extra space to allocate for a compressed variable. This extra space allows the variable to expand when you write additional records to the variable. If you do not specify this room for growth, the library has to move the variable to the end of the file when the size expands and the space at the original location of the variable becomes wasted space.

Examples

Create a CDF, create a variable, set the compression of the variable, and then set the reserve percent for the variable. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Set the compression of the variable.
cdflib.setVarCompression(cdfid,varNum,'GZIP_COMPRESSION',8);

% Set the compression reserver percentage
cdflib.setVarReservePercent(cdfid,varNum, 80);

cdflib.close(cdfid);
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarReservePercent`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarReservePercent` | `cdflib.setVarCompression` | `cdflib.getVarCompression`

Tutorials

- “Importing Common Data File Format (CDF) Files”
- “Exporting to Common Data File Format (CDF) Files”

Purpose

Specify number of cache buffers used for all variables

Syntax

`cdflib.setVarsCacheSize(cdfId,varNum,numBuffers)`

Description

`cdflib.setVarsCacheSize(cdfId,varNum,numBuffers)` specifies the number of cache buffers the CDF library uses for all the variables in the multifile format Common Data Format (CDF) file.

This function is not applicable to single-file CDFs. For more information about caching, see the *CDF User's Guide*.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varNum

Numeric value identifying a variable in the file. Variable identifiers (variable numbers) are zero-based.

numBuffers

Numeric value specifying the cache buffers.

Examples

Create a multifile CDF and specify the number of buffers used for all variables. To run this example, you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf')

% Set the format of the file to be multi-file
cdflib.setFormat(cdfid,'MULTI_FILE');

% Create a variable in the file
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Note how the library creates a separate file for the variable
ls your_file.*
```

```
your_file.cdf  your_file.z0

% Determine the number of cache buffers used with the variable
numBuf = cdflib.getVarCacheSize(cdfid,varNum)

numBuf =

     1

% Specify the number of cache buffers used by all variables in CDF.
cdflib.setVarsCacheSize(cdfid,6)

% Check the number of cache buffers used with the variable.
numBuf = cdflib.getVarCacheSize(cdfid,varNum)

numBuf =

     6

%  Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarsCacheSize`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarCacheSize` | `cdflib.setVarCacheSize`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Specify how variable handles sparse records

Syntax

`cdflib.getVarSparseRecords(cdfId,varNum,stype)`

Description

`cdflib.getVarSparseRecords(cdfId,varNum,stype)` specifies the sparse records type of a variable in a Common Data Format (CDF) file.

Input Arguments

cdfId

Identifier of a CDF file, returned by a call to `cdflib.create` or `cdflib.open`.

varNum

Number that identifies the variable to be set. Variable numbers are zero-based.

stype

One of the following text strings, or its numeric equivalent, that specifies how the variable handles sparse records.

Text String	Description
'NO_SPARSERECORDS'	No sparse records
'PAD_SPARSERECORDS'	For sparse records, the library uses the variable's pad value when reading values from a missing record.
'PREV_SPARSERECORDS'	For sparse records, the library uses values from the previous existing record when reading values from a missing record. If there is no previous existing record, the library uses the variable's pad value.

To get the numeric equivalent of these text string constants, use the `cdflib.getConstantValue` function.

Examples

Open a multifile CDF and close a variable.

Create a CDF, create a variable, and set the sparse records type of the variable. To run this example you must be in a writable folder.

```
cdfid = cdflib.create('your_file.cdf');

% Create a variable in the file.
varNum = cdflib.createVar(cdfid,'Time','cdf_int1',1,[],true,[]);

% Set the sparse records type of the variable
cdflib.setVarSparseRecords(cdfid,varNum,'PAD_SPARSERECORDS');

% Check the sparse records type of the variable
stype = cdflib.getVarSparseRecords(cdfid,varNum)

stype =

PAD_SPARSERECORDS


%Clean up
cdflib.delete(cdfid);

clear cdfid
```

References

This function corresponds to the CDF library C API routine `CDFsetzVarSparseRecords`.

To use this function, you must be familiar with the CDF C interface. Read the CDF documentation at the `CDF Web site`.

For copyright information, see the `cdfcopyright.txt` file.

See Also

`cdflib.getVarSparseRecords` | `cdflib.getConstantValue`

Tutorials

- "Importing Common Data File Format (CDF) Files"
- "Exporting to Common Data File Format (CDF) Files"

Purpose

Read data from Common Data Format (CDF) file

Syntax

```
data = cdfread(filename)
data = cdfread(filename, param1, val1, param2, val2, ...)
[data, info] = cdfread(filename, ...)
```

Description

`data = cdfread(filename)` reads all the data from the Common Data Format (CDF) file specified in the string `filename`. CDF data sets typically contain a set of variables, of a specific data type, each with an associated set of records. The variable might represent time values with each record representing a specific time that an observation was recorded. `cdfread` returns all the data in a cell array where each column represents a variable and each row represents a record associated with a variable. If the variables have varying numbers of associated records, `cdfread` pads the rows to create a rectangular cell array, using pad values defined in the CDF file.

Note Because `cdfread` creates temporary files, the current working directory must be writeable.

`data = cdfread(filename, param1, val1, param2, val2, ...)` reads data from the file, where *param1*, *param2*, and so on, can be any of the parameters listed in the following table.

`[data, info] = cdfread(filename, ...)` returns details about the CDF file in the `info` structure.

Parameter	Value
'Records'	A vector specifying which records to read. Record numbers are zero-based. `cdfread` returns a cell array with the same number of rows as the number of records read and as many columns as there are variables.
'Variables'	A 1-by-*n* or *n*-by-1 cell array specifying the names of the variables to read from the file. *n* must be less than or equal to the total number of variables in the file. `cdfread` returns a cell array with the same number of columns as the number of variables read, and a row for each record read.
'Slices'	An *m*-by-3 array, where each row specifies where to start reading along a particular dimension of a variable, the skip interval to use on that dimension (every item, every other item, etc.), and the total number of values to read on that dimension. *m* must be less than or equal to the number of dimensions of the variable. If *m* is less than the total number of dimensions, `cdfread` reads every value from the unspecified dimensions (`[0 1 n]`, where *n* is the total number of elements in the dimension. Note: Because the `'Slices'` parameter describes how to process a single variable, it must be used in conjunction with the `'Variables'` parameter.
'ConvertEpochToDatenum'	A Boolean value that determines whether `cdfread` automatically converts CDF epoch data types to MATLAB serial date numbers. If set to `false` (the default), `cdfread` wraps epoch values in MATLAB `cdfepoch` objects. Note: For better performance when reading large data sets, set this parameter to `true`.
'CombineRecords'	A Boolean value that determines how `cdfread` returns the CDF data sets read from the file. If set to `false` (the default), `cdfread` stores the data in an *m*-by-*n* cell array, where *m* is the number of records and *n* is the number of variables requested. If set to `true`, `cdfread` combines

Parameter	Value
	all records for a particular variable into one cell in the output cell array. In this cell, `cdfread` stores scalar data as a column array. `cdfread` extends the dimensionality of nonscalar and string data. For example, instead of creating 1000 elements containing 20-by-30 arrays for each record, `cdfread` stores all the records in one cell as a 1000-by-20-by-30 array Note: If you use the `'Records'` parameter to specify which records to read, you cannot use the `'CombineRecords'` parameter. Note: When using the `'Variable'` parameter to read one variable, if the `'CombineRecords'` parameter is `true`, `cdfread` returns the data as an M-by-N numeric or character array; it does not put the data into a cell array.

Note To improve performance when working with large data files, use the `'ConvertEpochToDatenum'` and `'CombineRecords'` options.

Note To improve performance, turn off the file validation which the CDF library does by default when opening files. For more information, see `cdflib.setValidate`.

Examples

Read all the data from a CDF file.

```
data = cdfread('example.cdf');
```

Read the data from the variable `'Time'`.

```
data = cdfread('example.cdf', 'Variable', {'Time'});
```

Read the first value in the first dimension, the second value in the second dimension, the first and third values in the third dimension, and all values in the remaining dimension of the variable `'multidimensional'`.

```
data = cdfread('example.cdf', ...
               'Variable', {'multidimensional'}, ...
               'Slices', [0 1 1; 1 1 1; 0 2 2]);
```

This is similar to reading the whole variable into `data` and then using matrix indexing, as in the following.

```
data{1}(1, 2, [1 3], :)
```

Collapse the records from a data set and convert CDF epoch data types to MATLAB serial date numbers.

```
data = cdfread('example.cdf', ...
               'CombineRecords', true, ...
               'ConvertEpochToDatenum', true);
```

See Also

`cdfepoch` | `cdfinfo` | `cdflib.setValidate`

How To

- "Importing Common Data File Format (CDF) Files"

Purpose

Write data to Common Data Format (CDF) file

Note `cdfwrite` is not recommended. Use the `cdflib` low-level functions instead.

Syntax

```
cdfwrite(filename,variablelist)
cdfwrite(...,'PadValues',padvals)
cdfwrite(...,'GlobalAttributes',gattrib)
cdfwrite(..., 'VariableAttributes', vattrib)
cdfwrite(...,'WriteMode',mode)
cdfwrite(...,'Format',format)
```

Description

`cdfwrite(filename,variablelist)` writes out a Common Data Format (CDF) file, specified in `filename`. The `filename` input is a string enclosed in single quotes. The `variablelist` argument is a cell array of ordered pairs, each of which comprises a CDF variable name (a string) and the corresponding CDF variable value. To write out multiple records for a variable, put the values in a cell array where each element in the cell array represents a record.

Note Because `cdfwrite` creates temporary files, both the destination directory for the file and the current working directory must be writeable.

`cdfwrite(...,'PadValues',padvals)` writes out pad values for given variable names. `padvals` is a cell array of ordered pairs, each of which comprises a variable name (a string) and a corresponding pad value. Pad values are the default values associated with the variable when an out-of-bounds record is accessed. Variable names that appear in `padvals` must appear in `variablelist`.

`cdfwrite(...,'GlobalAttributes',gattrib)` writes the structure `gattrib` as global metadata for the CDF file. Each field of the structure

is the name of a global attribute. The value of each field contains the value of the attribute. To write out multiple values for an attribute, put the values in a cell array where each element in the cell array represents a record.

Note To specify a global attribute name that is invalid in your MATLAB application, create a field called `'CDFAttributeRename'` in the attribute structure. The value of this field must have a value that is a cell array of ordered pairs. The ordered pair consists of the name of the original attribute, as listed in the `GlobalAttributes` structure, and the corresponding name of the attribute to be written to the CDF file.

`cdfwrite(..., 'VariableAttributes', vattrib)` writes the structure `vattrib` as variable metadata for the CDF. Each field of the struct is the name of a variable attribute. The value of each field should be an M-by-2 cell array where M is the number of variables with attributes. The first element in the cell array should be the name of the variable and the second element should be the value of the attribute for that variable.

Note To specify a variable attribute name that is illegal in MATLAB, create a field called `'CDFAttributeRename'` in the attribute structure. The value of this field must have a value that is a cell array of ordered pairs. The ordered pair consists of the name of the original attribute, as listed in the `VariableAttributes` struct, and the corresponding name of the attribute to be written to the CDF file. If you are specifying a variable attribute of a CDF variable that you are renaming, the name of the variable in the `VariableAttributes` structure must be the same as the renamed variable.

`cdfwrite(...,'WriteMode',`*`mode`*`)`, where *`mode`* is either `'overwrite'` or `'append'`, indicates whether or not the specified variables should be

appended to the CDF file if the file already exists. By default, `cdfwrite` overwrites existing variables and attributes.

`cdfwrite(...,'Format',`*`format`*`)`, where *`format`* is either `'multifile'` or `'singlefile'`, indicates whether or not the data is written out as a multifile CDF. In a multifile CDF, each variable is stored in a separate file with the name `*.vN`, where `N` is the number of the variable that is written out to the CDF. By default, `cdfwrite` writes out a single file CDF. When `'WriteMode'` is set to `'Append'`, the `'Format'` option is ignored, and the format of the preexisting CDF is used.

Examples

Write out a file `'example.cdf'` containing a variable `'Longitude'` with the value `[0:360]`.

```
cdfwrite('example', {'Longitude', 0:360});
```

Write out a file `'example.cdf'` containing variables `'Longitude'` and `'Latitude'` with the variable `'Latitude'` having a pad value of 10 for all out-of-bounds records that are accessed.

```
cdfwrite('example', {'Longitude', 0:360, 'Latitude', 10:20}, ...
        'PadValues', {'Latitude', 10});
```

Write out a file `'example.cdf'`, containing a variable `'Longitude'` with the value `[0:360]`, and with a variable attribute of `'validmin'` with the value 10.

```
varAttribStruct.validmin = {'Longitude' [10]};
cdfwrite('example', {'Longitude' 0:360}, 'VariableAttributes', ...
        varAttribStruct);
```

See Also

`cdfread` | `cdfinfo` | `cdfepoch`

ceil

Purpose

Round toward positive infinity

Syntax

```
B = ceil(A)
```

Description

`B = ceil(A)` rounds the elements of `A` to the nearest integers greater than or equal to `A`. For complex `A`, the imaginary and real parts are rounded independently.

Examples

```
a = [-1.9, -0.2, 3.4, 5.6, 7, 2.4+3.6i]

a =
  Columns 1 through 4
  -1.9000        -0.2000        3.4000        5.6000

  Columns 5 through 6
   7.0000        2.4000 + 3.6000i

ceil(a)

ans =
  Columns 1 through 4
  -1.0000        0              4.0000        6.0000

  Columns 5 through 6
   7.0000        3.0000 + 4.0000i
```

See Also

`fix` | `floor` | `round`

Purpose

Create cell array

Syntax

```
C = cell(dim)
C = cell(dim1,...,dimN)
D = cell(obj)
```

Description

`C = cell(dim)` creates a cell array of empty matrices. If `dim` is a scalar, `C` is `dim`-by-`dim`. If `dim` is a vector, `C` is `dim(1)`-by-...-`dim(N)`, where `N` is the number of elements of `dim`.

`C = cell(dim1,...,dimN)` creates cell array `C`, where `C` is `dim1`-by-...-`dimN`.

`D = cell(obj)` converts a Java array or .NET array of `System.String` or `System.Object` into a MATLAB cell array.

Tips

Creating an empty array with the `cell` function, such as

```
C = cell(3,4,2);
```

is exactly equivalent to assigning an empty array to the last index of a new cell array:

```
C{3,4,2} = [];
```

Input Arguments

dim

Scalar integer or vector of integers that specifies the dimensions of cell array `C`.

dim1,...,dimN

Scalar integers that specify the dimensions of `C`.

obj

One of the following:

- Java array or object
- .NET array of type `System.String` or `System.Object`

Output Arguments

C

Cell array. Each cell contains an empty, 0-by-0 array of type `double`.

D

Cell array. Each cell contains a MATLAB type closest to the Java or .NET type. For more information, see:

- “Conversion of Java Return Types”
- “.NET Type to MATLAB Type Mapping”

Examples

Create an empty 3-by-4-by-2 cell array.

```
mycell = cell(3,4,2);
```

Create a cell array that is the same size as `mycell`, created in the previous example.

```
similar = cell(size(mycell));
```

Convert an array of `java.lang.String` objects into a MATLAB cell array.

```
strArray = java_array('java.lang.String', 3);
strArray(1) = java.lang.String('one');
strArray(2) = java.lang.String('two');
strArray(3) = java.lang.String('three');

cellArray = cell(strArray)
```

This code returns

```
cellArray =
    'one'
    'two'
```

```
    'three'
```

Create a cell array of folders in the `c:\work` folder, using the .NET Framework `System.IO.Directory` class :

```
myList = cell(System.IO.Directory.GetDirectories('c:\work'));
celldisp(myList)
```

See Also

`num2cell` | `ones` | `rand` | `randn` | `zeros`

How To

- "Access Data in a Cell Array"

cell2mat

Purpose Convert cell array to numeric array

Syntax

```
A= cell2mat(C)
```

Description

`A= cell2mat(C)` converts cell array `C` with contents of the same data type into a single array, `A`.

Input Arguments

C

Cell array, where each cell contains the same type of data. The `cell2mat` function accepts numeric or character data within cells of `C`, but not structs, objects, or nested cells.

The contents of `C` must support concatenation into a hyperrectangle. Otherwise, the results are undefined. For example, the contents of cells in the same column must have the same number of columns, although they need not have the same number of rows.

10x25 10x25
20x25 20x25
30x25 30x25
cell2mat
60x50

Output Arguments

A

Array of the same type as the contents of the cells of `C`. The number of dimensions of `A` matches the highest number of dimensions of arrays within `C`.

Examples

Combine matrices in the four cells of cell array `c` into matrix `m`.

```
c = {[1],    [2, 3, 4];
     [5; 9], [6, 7, 8; 10, 11, 12]};

m = cell2mat(c)
```

`m` is a 3-by-4 matrix:

```
m =
     1     2     3     4
     5     6     7     8
     9    10    11    12
```

See Also

`mat2cell` | `num2cell`

cell2struct

Purpose

Convert cell array to structure array

Syntax

`structArray = cell2struct(cellArray, fields, dim)`

Description

`structArray = cell2struct(cellArray, fields, dim)` creates a structure array, `structArray`, from the information contained within cell array `cellArray`.

The `fields` argument specifies field names for the structure array. This argument is an array of strings or a cell array of strings.

The `dim` argument tells MATLAB which axis of the cell array to use in creating the structure array. Use a numeric `double` to specify `dim`.

To create a structure array with fields derived from `N` rows of a cell array, specify `N` field names in the `fields` argument, and the number 1 in the `dim` argument. To create a structure array with fields derived from `M` columns of a cell array, specify `M` field names in the `fields` argument and the number 2 in the `dim` argument.

The `structArray` output is a structure array with `N` fields, where `N` is equal to the number of fields in the `fields` input argument. The number of fields in the resulting structure must equal the number of cells along dimension `dim` that you want to convert.

Examples

Create the following table for use with the examples in this section. The table lists information about the employees of a small Engineering company. Reading the table by rows shows the names of employees by department. Reading the table by columns shows the number of years each employee has worked at the company.

	5 Years	10 Years	15 Years
Development	Lee, Reed, Hill	Dean, Frye	Lane, Fox, King
Sales	Howe, Burns	Kirby, Ford	Hall
Management	Price	Clark, Shea	Sims

	5 Years	10 Years	15 Years
Quality	Bates, Gray	Nash	Kay, Chase
Documentation	Lloyd, Young	Ryan, Hart, Roy	Marsh

Enter the following commands to create the initial cell array `employees`:

```
devel = {{'Lee','Reed','Hill'}, {'Dean','Frye'}, ...
   {'Lane','Fox','King'}};
sales = {{'Howe','Burns'}, {'Kirby','Ford'}, {'Hall'}};
mgmt = {{'Price'}, {'Clark','Shea'}, {'Sims'}};
qual = {{'Bates','Gray'}, {'Nash'}, {'Kay','Chase'}};
docu = {{'Lloyd','Young'}, {'Ryan','Hart','Roy'}, {'Marsh'}};

employees = [devel; sales; mgmt; qual; docu]
employees =
    {1x3 cell}    {1x2 cell}    {1x3 cell}
    {1x2 cell}    {1x2 cell}    {1x1 cell}
    {1x1 cell}    {1x2 cell}    {1x1 cell}
    {1x2 cell}    {1x2 cell}    {1x2 cell}
    {1x2 cell}    {1x2 cell}    {1x1 cell}
```

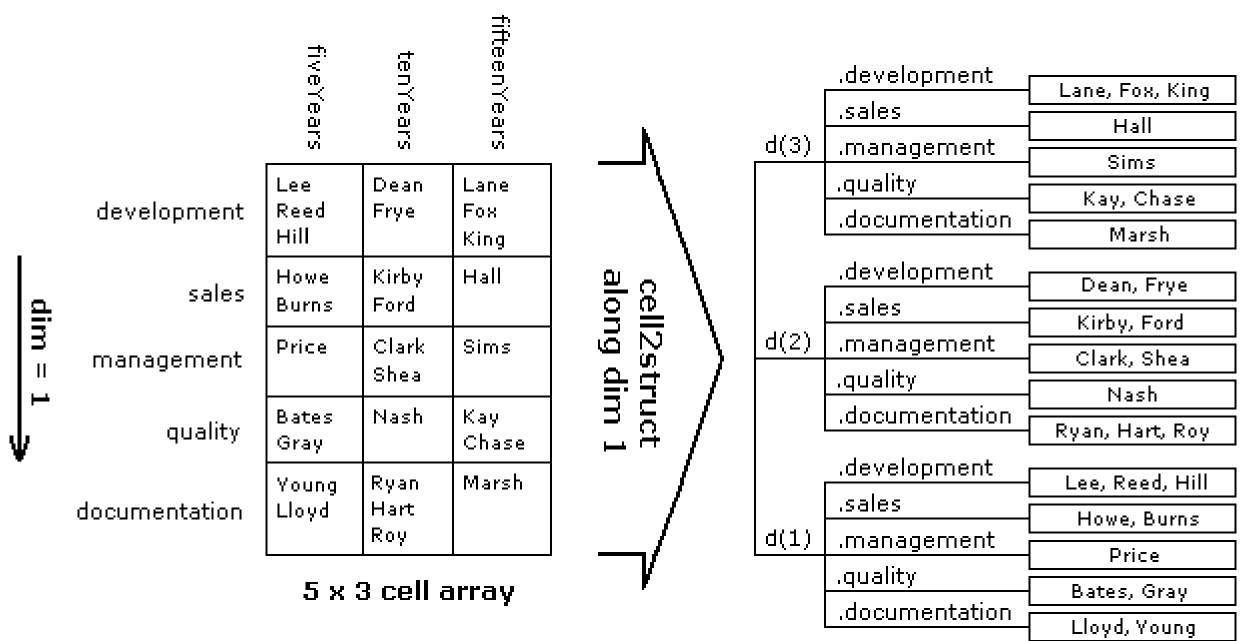
This is the resulting cell array:

	fiveYears	tenYears	fifteenYears
development	Lee Reed Hill	Dean Frye	Lane Fox King
sales	Howe Burns	Kirby Ford	Hall
management	Price	Clark Shea	Sims
quality	Bates Gray	Nash	Kay Chase
documentation	Young Lloyd	Ryan Hart Roy	Marsh

5 x 3 cell array

Convert the cell array to a struct along dimension 1:

1. Convert the 5-by-3 cell array along its first dimension to construct a 3-by-1 struct array with 5 fields. Each of the rows along dimension 1 of the cell array becomes a field in the struct array:



Traversing the first (i.e., vertical) dimension, there are 5 rows with row headings that read as follows:

```
rowHeadings = {'development', 'sales', 'management', ...
   'quality', 'documentation'};
```

2 Convert the cell array to a struct array, `depts`, in reference to this dimension:

```
depts = cell2struct(employees, rowHeadings, 1)
depts =
3x1 struct array with fields:
    development
    sales
```

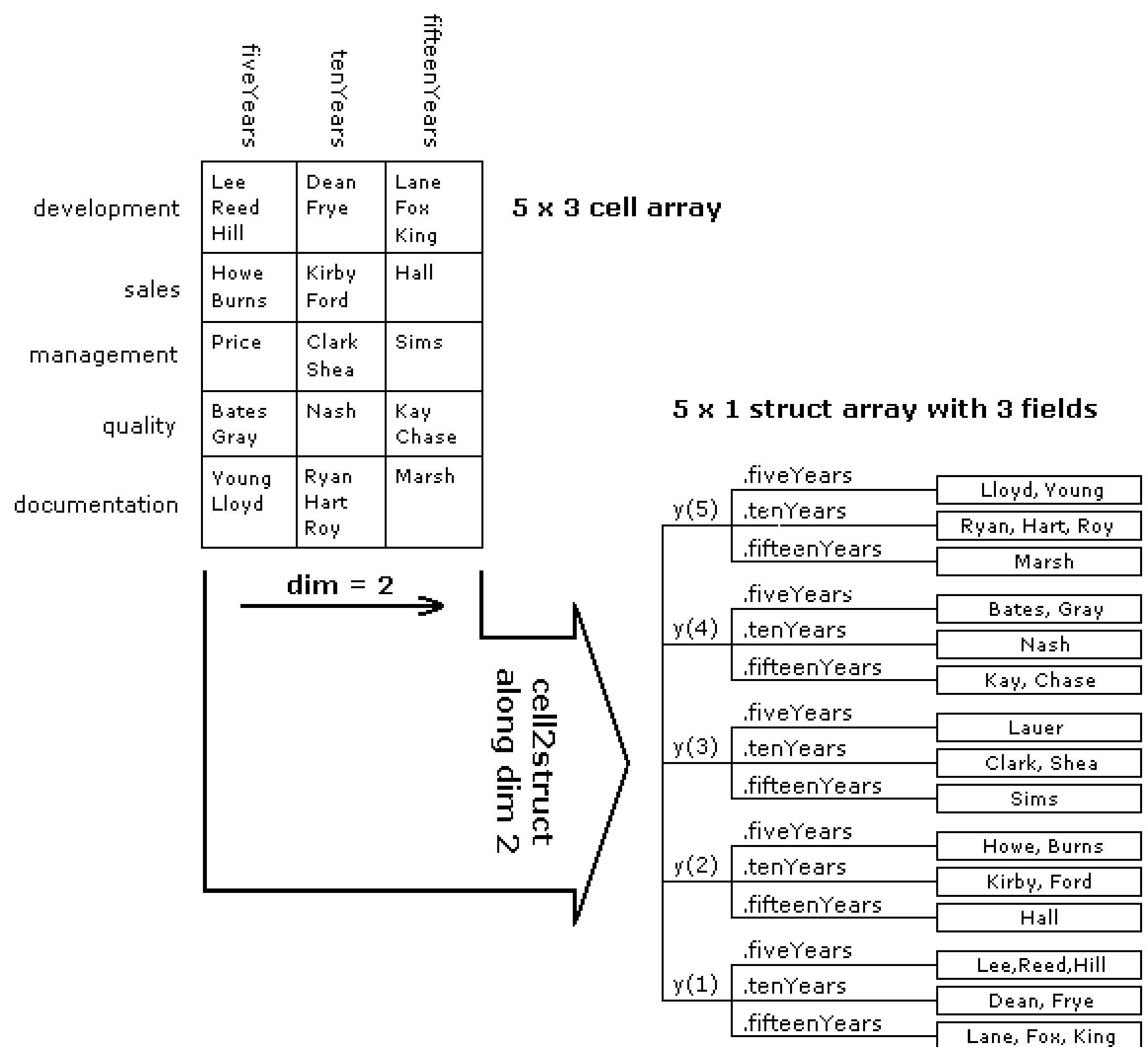
```
management
quality
documentation
```

3 Use this row-oriented structure to find the names of the Development staff who have been with the company for up to 10 years:

```
depts(1:2).development
ans =
    'Lee'    'Reed'    'Hill'
ans =
    'Dean'    'Frye'
```

Convert the same cell array to a struct along dimension 2:

1 Convert the 5-by-3 cell array along its second dimension to construct a 5-by-1 struct array with 3 fields. Each of the columns along dimension 2 of the cell array becomes a field in the struct array:


fiveYears
tenYears
fifteenYears
development
sales
management
quality
documentation
Lee
Reed
Hill
Dean
Frye
Lane
Fox
King
Howe
Burns
Kirby
Ford
Hall
Price
Clark
Shea
Sims
Bates
Gray
Nash
Kay
Chase
Young
Lloyd
Ryan
Hart
Roy
Marsh
5 x 3 cell array
dim = 2
cell2struct
along dim 2
5 x 1 struct array with 3 fields
y(5)
y(4)
y(3)
y(2)
y(1)
.fiveYears
.tenYears
.fifteenYears
Lloyd, Young
Ryan, Hart, Roy
Marsh
Bates, Gray
Nash
Kay, Chase
Lauer
Clark, Shea
Sims
Howe, Burns
Kirby, Ford
Hall
Lee,Reed,Hill
Dean, Frye
Lane, Fox, King

2 Traverse the cell array along the second (or horizontal) dimension. The column headings become fields of the resulting structure:

```
colHeadings = {'fiveYears' 'tenYears' 'fifteenYears'};

years = cell2struct(employees, colHeadings, 2)
years =
5x1 struct array with fields:
    fiveYears
    tenYears
    fifteenYears
```

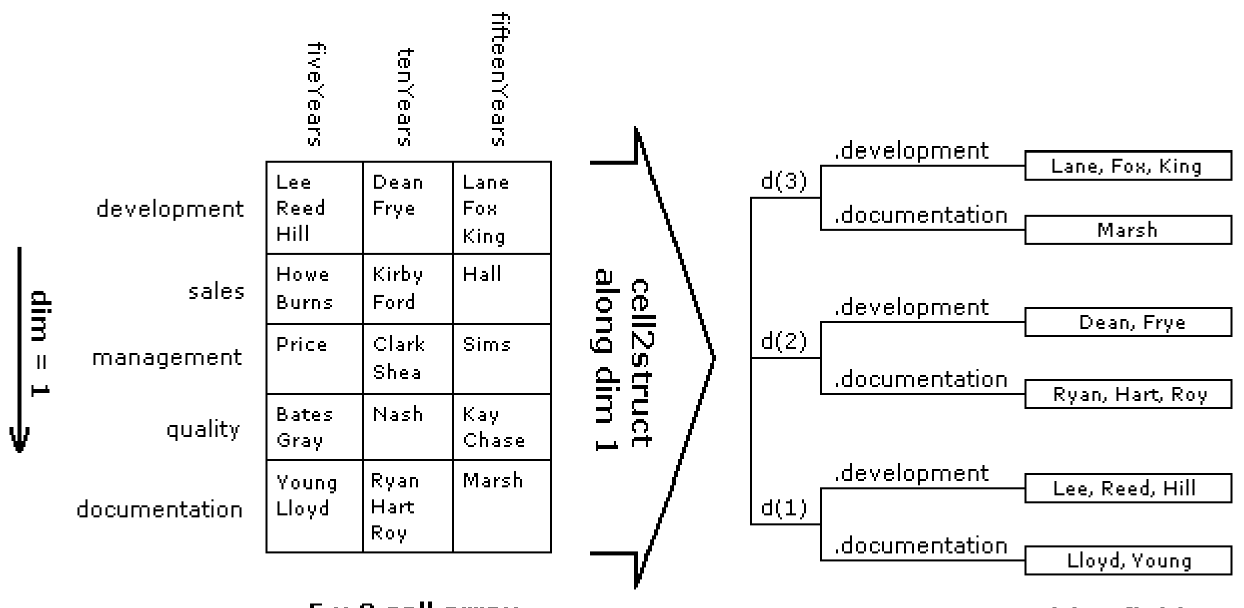
3 Using the column-oriented structure, show how many employees from the Sales and Documentation departments have worked for the company for at least 5 years:

```
[~, sales_5years, ~, ~, docu_5years] = years.fiveYears
sales_5years =
    'Howe'    'Burns'
docu_5years =
    'Lloyd'    'Young'
```

Convert only part of the cell array to a struct:

1 Convert only the first and last rows of the cell array. This results in a 3-by-1 struct array with 2 fields:

```
rowHeadings = {'development', 'documentation'};

depts = cell2struct(employees([1,5],:), rowHeadings, 1)
depts =
3x1 struct array with fields:
    development
    documentation
```



2 Display those employees who belong to these departments for all three periods of time:

```
for k=1:3
   depts(k,:)
end

ans =
      development: {'Lee'   'Reed'   'Hill'}
    documentation: {'Lloyd'   'Young'}
ans =
      development: {'Dean'   'Frye'}
    documentation: {'Ryan'   'Hart'   'Roy'}
ans =
      development: {'Lane'   'Fox'   'King'}
```

```
documentation: {'Marsh'}
```

See Also

`struct2cell` | `cell` | `iscell` | `struct` | `isstruct` | `fieldnames`

How To

- dynamic field names

celldisp

Purpose

Cell array contents

Syntax

```
celldisp(C)
celldisp(C, name)
```

Description

`celldisp(C)` recursively displays the contents of a cell array.

`celldisp(C, name)` uses the string *name* for the display instead of the name of the first input (or `ans`).

Examples

Use `celldisp` to display the contents of a 2-by-3 cell array:

```
C = {[1 2] 'Tony' 3+4i; [1 2;3 4] -5 'abc'};
celldisp(C)

C{1,1} =
     1     2

C{2,1} =
     1     2
     3     4

C{1,2} =
Tony

C{2,2} =
    -5

C{1,3} =
   3.0000+ 4.0000i

C{2,3} =
abc
```

See Also

`cellplot`

How To

- "Exporting a Cell Array to a Text File"

Purpose

Apply function to each cell in cell array

Syntax

```
[A1,...,Am] = cellfun(func,C1,...,Cn)
[A1,...,Am] = cellfun(func,C1,...,Cn,Name,Value)
```

Description

`[A1,...,Am] = cellfun(func,C1,...,Cn)` calls the function specified by function handle `func` and passes elements from cell arrays `C1,...,Cn`, where `n` is the number of inputs to function `func`. Output arrays `A1,...,Am`, where `m` is the number of outputs from function `func`, contain the combined outputs from the function calls. The `i`th iteration corresponds to the syntax `[A1(i),...,Am(i)] = func(C{i},...,Cn{i})`. The `cellfun` function does not perform the calls to function `func` in a specific order.

`[A1,...,Am] = cellfun(func,C1,...,Cn,Name,Value)` calls function `func` with additional options specified by one or more `Name,Value` pair arguments. Possible values for `Name` are `'UniformOutput'` or `'ErrorHandler'`.

Input Arguments

func

Handle to a function that accepts `n` input arguments and returns `m` output arguments.

If function `func` corresponds to more than one function file (that is, if `func` represents a set of overloaded functions), MATLAB determines which function to call based on the class of the input arguments.

Backward Compatibility

`cellfun` accepts function name strings for function `func`, rather than a function handle, for these function names: `isempty`, `islogical`, `isreal`, `length`, `ndims`, `prodofsize`, `size`, `isclass`. Enclose the function name in single quotes.

If you specify a function name string rather than a function handle:

- `cellfun` does not call any overloaded versions of the function.

- The `size` and `isclass` functions require additional inputs to the `cellfun` function:

  `A = cellfun('size', C, k)` returns the size along the `k`th dimension of each element of `C`.

  `A = cellfun('isclass', C, classname)` returns logical `1` (`true`) for each element of `C` that matches the `classname` string. This syntax returns logical 0 (`false`) for objects that are a subclass of `classname`.

C1,...,Cn

Cell arrays that contain the `n` inputs required for function `func`. Each cell array must have the same dimensions.

Name-Value Pair Arguments

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

'UniformOutput'

Logical value, as follows:

`true (1)`	Indicates that for all inputs, each output from function `func` is a cell array or a scalar value that is always of the same type. The `cellfun` function combines the outputs in arrays `A1,...,Am`, where `m` is the number of function outputs. Each output array is of the same type as the individual function outputs.
`false (0)`	Requests that the `cellfun` function combine the outputs into cell arrays `A1,...,Am`. The outputs of function `func` can be of any size or type.

Default: `true`

'ErrorHandler'

Handle to a function that catches any errors that occur when MATLAB attempts to execute function `func`. Define this function so that it rethrows the error or returns valid outputs for function `func`.

MATLAB calls the specified error-handling function with two input arguments:

- A structure with these fields:

`identifier`	Error identifier.
`message`	Error message text.
`index`	Linear index corresponding to the element of the input cell array at the time of the error.

- The set of input arguments to function `func` at the time of the error.

Output Arguments

A1,...,Am

Arrays that collect the `m` outputs from function `func`. Each array `A` is the same size as each of the inputs `C1,...,Cn`.

Function `func` can return output arguments of different classes. However, if `UniformOutput` is `true` (the default):

- The individual outputs from function `func` must be scalar values (numeric, logical, character, or structure) or cell arrays.
- The class of a particular output argument must be the same for each set of inputs. The class of the corresponding output array is the same as the class of the outputs from function `func`.

Examples

Compute the mean of each vector in cell array `C`.

```
C = {1:10, [2; 4; 6], []};

averages = cellfun(@mean, C)
```

This code returns

```
averages =
    5.5000    4.0000       NaN
```

Compute the size of each array in `C`, created in the previous example.

```
[nrows, ncols] = cellfun(@size, C)
```

This code returns

```
nrows =
     1     3     0
ncols =
    10     1     0
```

Create a cell array that contains strings, and abbreviate those strings to the first three characters. Because the output strings are nonscalar, set `UniformOutput` to `false`.

```
days = {'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday'};

abbrev = cellfun(@(x) x(1:3), days, 'UniformOutput', false)
```

The syntax `@(x)` creates an anonymous function. This code returns

```
abbrev =
    'Mon'    'Tue'    'Wed'    'Thu'    'Fri'
```

Compute the covariance between arrays in two cell arrays `C` and `D`. Because the covariance output is nonscalar, set `UniformOutput` to `false`.

```
c1 = rand(5,1);  c2 = rand(10,1);  c3 = rand(15,1);
d1 = rand(5,1);  d2 = rand(10,1);  d3 = rand(15,1);
```

```
C = {c1, c2, c3};
D = {d1, d2, d3};

covCD = cellfun(@cov, C, D, 'UniformOutput', false)
```

This code returns

```
covCD =
    [2x2 double]    [2x2 double]    [2x2 double]
```

Define and call a custom error handling function.

```
function result = errorfun(S, varargin)
   warning(S.identifier, S.message);
   result = NaN;
end

A = {rand(3)};
B = {rand(5)};
AgtB = cellfun(@(x,y) x > y, A, B, 'ErrorHandler', @errorfun, ...
               'UniformOutput', false)
```

See Also

arrayfun | spfun | function_handle | cell2mat

Tutorials

- “Anonymous Functions”

Purpose

Graphically display structure of cell array

Syntax

```
cellplot(c)
cellplot(c, 'legend')
handles = cellplot(c)
```

Description

`cellplot(c)` displays a figure window that graphically represents the contents of `c`. Filled rectangles represent elements of vectors and arrays, while scalars and short text strings are displayed as text.

`cellplot(c, 'legend')` places a colorbar next to the plot labelled to identify the data types in `c`.

`handles = cellplot(c)` displays a figure window and returns a vector of surface handles.

Limitations

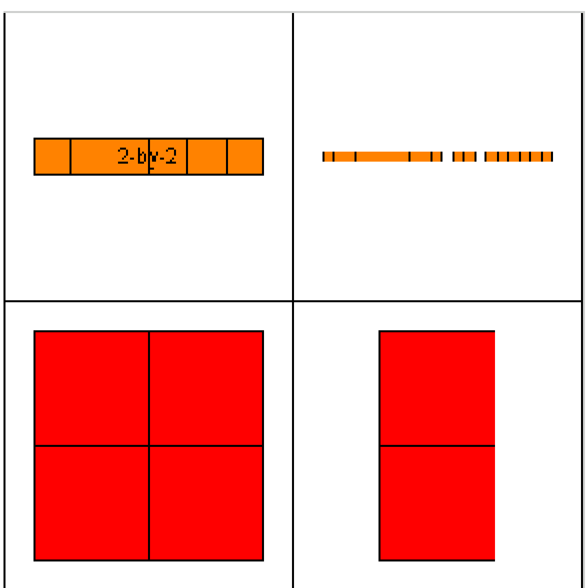
The `cellplot` function can display only two-dimensional cell arrays.

Examples

Consider a 2-by-2 cell array containing a matrix, a vector, and two text strings:

```
c{1,1} = '2-by-2';
c{1,2} = 'eigenvalues of eye(2)';
c{2,1} = eye(2);
c{2,2} = eig(eye(2));
```

The command `cellplot(c)` produces



See Also

`celldisp`

How To

- "Editing Plots"
- "Exporting a Cell Array to a Text File"

Purpose

Create cell array of strings from character array

Syntax

```
c = cellstr(S)
```

Description

`c = cellstr(S)` places each row of the character array `S` into separate cells of `c`. Any trailing spaces in the rows of `S` are removed.

Use the `char` function to convert back to a string matrix.

Examples

Given the string matrix

```
S = ['abc '; 'defg'; 'hi  ']

S =
    abc
    defg
    hi

whos S
  Name      Size         Bytes  Class
  S         3x4             24  char array
```

The following command returns a 3-by-1 cell array.

```
c = cellstr(S)

c =
    'abc'
    'defg'
    'hi'

whos c
  Name      Size         Bytes  Class
  c         3x1            198  cell array
```

See Also

`iscellstr` | `strings` | `char` | `isstrprop` | `strsplit`

Purpose

Conjugate gradients squared method

Syntax

```
x = cgs(A,b)
cgs(A,b,tol)
cgs(A,b,tol,maxit)
cgs(A,b,tol,maxit,M)
cgs(A,b,tol,maxit,M1,M2)
cgs(A,b,tol,maxit,M1,M2,x0)
[x,flag] = cgs(A,b,...)
[x,flag,relres] = cgs(A,b,...)
[x,flag,relres,iter] = cgs(A,b,...)
[x,flag,relres,iter,resvec] = cgs(A,b,...)
```

Description

`x = cgs(A,b)` attempts to solve the system of linear equations `A*x = b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. You can specify `A` as a function handle, `afun`, such that `afun(x)` returns `A*x`.

"Parameterizing Functions" explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `cgs` converges, a message to that effect is displayed. If `cgs` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`cgs(A,b,tol)` specifies the tolerance of the method, `tol`. If `tol` is `[]`, then `cgs` uses the default, `1e-6`.

`cgs(A,b,tol,maxit)` specifies the maximum number of iterations, `maxit`. If `maxit` is `[]` then `cgs` uses the default, `min(n,20)`.

`cgs(A,b,tol,maxit,M)` and `cgs(A,b,tol,maxit,M1,M2)` use the preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `cgs` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x)` returns `M\x`.

`cgs(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess `x0`. If `x0` is `[]`, then `cgs` uses the default, an all-zero vector.

`[x,flag] = cgs(A,b,...)` returns a solution `x` and a flag that describes the convergence of `cgs`.

Flag	Convergence
0	`cgs` converged to the desired tolerance `tol` within `maxit`iterations.
1	`cgs` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	`cgs` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `cgs` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = cgs(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, then `relres <= tol`.

`[x,flag,relres,iter] = cgs(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = cgs(A,b,...)` also returns a vector of the residual norms at each iteration, including `norm(b-A*x0)`.

Examples

Using cgs with a Matrix Input

```
A = gallery('wilk',21);
b = sum(A,2);
tol = 1e-12;  maxit = 15;
M1 = diag([10:-1:1 1 1:10]);
x = cgs(A,b,tol,maxit,M1);
```

displays the message

```
cgs converged at iteration 13 to a solution with
relative residual 2.4e-016.
```

Using cgs with a Function Handle

This example replaces the matrix `A` in the previous example with a handle to a matrix-vector product function `afun`, and the preconditioner `M1` with a handle to a backsolve function `mfun`. The example is contained in the file `run_cgs` that

- Calls `cgs` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_cgs` are available to `afun` and `myfun`.

The following shows the code for `run_cgs`:

```
function x1 = run_cgs
n = 21;
b = afun(ones(n,1));
tol = 1e-12;  maxit = 15;
x1 = cgs(@afun,b,tol,maxit,@mfun);

    function y = afun(x)
        y = [0; x(1:n-1)] + ...
              [((n-1)/2:-1:0)'; (1:(n-1)/2)'].*x + ...
              [x(2:n); 0];
    end

    function y = mfun(r)
        y = r ./ [((n-1)/2:-1:1)'; 1; (1:(n-1)/2)'];
    end
end
```

When you enter

```
x1 = run_cgs
```

MATLAB software returns

```
cgs converged at iteration 13 to a solution with
relative residual 2.4e-016.
```

Using a Preconditioner

This example demonstrates the use of a preconditioner.

1

Load `west0479`, a real 479-by-479 nonsymmetric sparse matrix:

```
load west0479;
A = west0479;
```

2

Define `b` so that the true solution is a vector of all ones:

```
b = full(sum(A,2));
```

3

Set the tolerance and maximum number of iterations:

```
tol = 1e-12; maxit = 20;
```

4

Use `cgs` to find a solution at the requested tolerance and number of iterations:

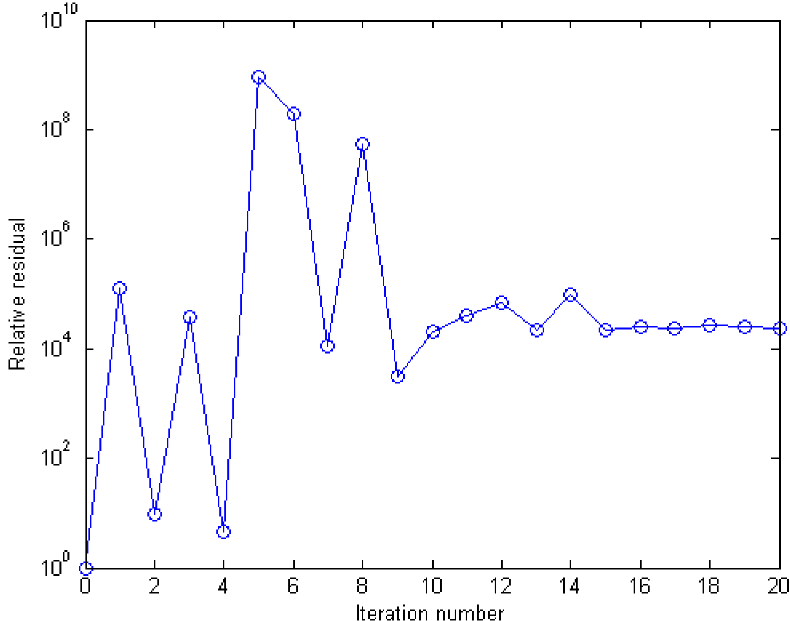
```
[x0,fl0,rr0,it0,rv0] = cgs(A,b,tol,maxit);
```

`fl0` is 1 because `cgs` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. In fact, the behavior of `cgs` is so poor that the initial guess (`x0 = zeros(size(A,2),1)`) is the best solution and is returned as indicated by `it0 = 0`. MATLAB stores the residual history in `rv0`.

5

Plot the behavior of cgs:

```
semilogy(0:maxit,rv0/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



The plot shows that the solution does not converge. You can use a preconditioner to improve the outcome.

6

Create a preconditioner with ilu, since A is nonsymmetric:

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));
Error using ilu
There is a pivot equal to zero.  Consider decreasing the
```

```
drop tolerance or consider using the 'udiag' option.
```

MATLAB cannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

7

You can try again with a reduced drop tolerance, as indicated by the error message:

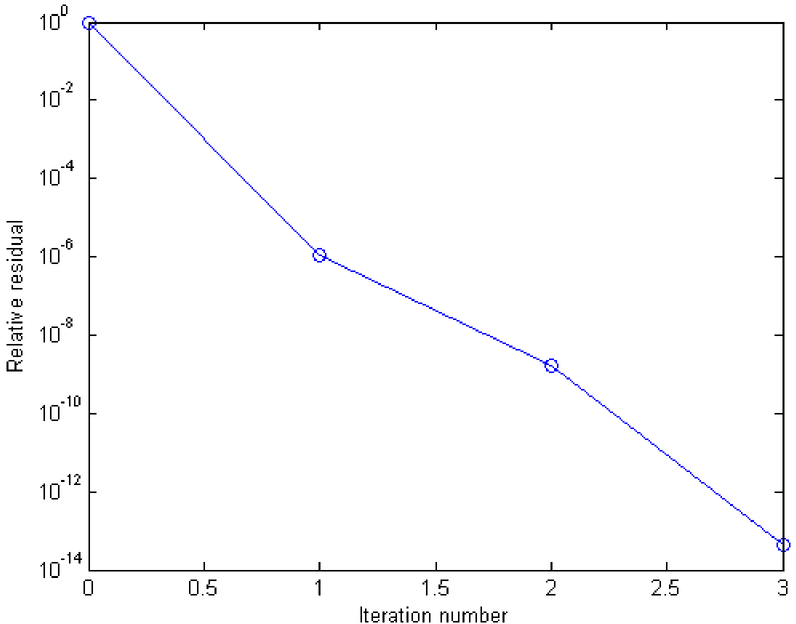
```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = cgs(A,b,tol,maxit,L,U);
```

`fl1` is 0 because `cgs` drives the relative residual to `4.3851e-014` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the third iteration (the value of `it1`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output `rv1(1)` is `norm(b)`and the output `rv1(14)` is `norm(b-A*x2)`.

8

You can follow the progress of `cgs` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0)

```
semilogy(0:it1,rv1/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Sonneveld, Peter, "CGS: A fast Lanczos-type solver for nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, January 1989, Vol. 10, No. 1, pp. 36–52.

See Also

`bicg` | `bicgstab` | `gmres` | `lsqr` | `luinc` | `minres` | `pcg` | `qmr` | `symmlq` | `function_handle` | `mldivide`

Purpose

Convert to character array (string)

Syntax

```
S = char(X)
S = char(C)
S = char(T1,T2,...,TN)
```

Description

`S = char(X)` converts array `X` of nonnegative integer codes into a character array. Valid codes range from 0 to 65535, where codes 0 through 127 correspond to 7-bit ASCII characters. The characters that MATLAB can process (other than 7-bit ASCII characters) depend upon your current locale setting. To convert characters into a numeric array, use the `double` function.

`S = char(C)`, when `C` is a cell array of strings, places each element of `C` into the rows of the character array `s`. Use `cellstr` to convert back.

`S = char(T1,T2,...,TN)` forms the character array `S` containing the text strings `T1`, `T2`, `...`, `TN` as rows, automatically padding each string with blanks to form a valid matrix. Each text parameter, `Ti`, can itself be a character array. This allows the creation of arbitrarily large character arrays. Empty strings are significant.

Examples

Convert Integers to Characters

Create a 3-by-32 array of the printable ASCII characters.

```
asc = char(reshape(32:127,32,3)')
```

```
asc =

 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~
```

See Also

`ischar` | `isletter` | `isspace` | `isstrprop` | `cellstr` | `iscellstr` | `get` | `set` | `strings` | `text`

Tutorials

- “How the MATLAB Process Uses Locale Settings”

checkcode

Purpose

Check MATLAB code files for possible problems

Alternatives

For information on using the graphical user interface for checking code, see “Check Code for Errors and Warnings”.

Syntax

```
checkcode('filename')
checkcode('filename','-config=settings.txt')
checkcode('filename','-config=factory')
inform=checkcode('filename','-struct')
msg=checkcode('filename','-string')
[inform,filepaths]=checkcode('filename')
inform=checkcode('filename','-id')
inform=checkcode('filename','-fullpath')
inform=checkcode('filename','-notok')
checkcode('filename','-cyc')
checkcode('filename','-codegen')
checkcode('filename','-eml')
```

Description

`checkcode('filename')` displays messages, sometimes referred to as Code Analyzer messages, about `filename`, where the message reports potential problems and opportunities for code improvement. The line number in the message is a hyperlink that opens the file in the Editor, scrolled to that line. If `filename` is a cell array, information is displayed for each file. For `checkcode(F1,F2,F3,...)`, where each input is a character array, MATLAB software displays information about each input file name. You cannot combine cell arrays and character arrays of file names. Note that the exact text of the `checkcode` messages is subject to some change between versions.

`checkcode('filename','-config=settings.txt')` overrides the default active settings file with the settings that enable or suppress messages as indicated in the specified `settings.txt` file.

Note If used, you must specify the full path to the `settings.txt` file specified with the `-config` option.

For information about creating a `settings.txt` file, see "Save and Reuse Code Analyzer Message Settings". If you specify an invalid file, `checkcode` returns a message indicating that it cannot open or read the file you specified. In that case, `checkcode` uses the factory default settings.

`checkcode('filename','-config=factory')` ignores all settings files and uses the factory default preference settings.

`inform=checkcode('filename','-struct')` returns the information in a structure array whose length is the number of messages found. The structure has the fields that follow.

Field	Description
`message`	Message describing the suspicious construct that code analysis caught.
`line`	Vector of file line numbers to which the message refers.
`column`	Two-column array of file columns (column extents) to which the message applies. The first column of the array specifies the column in the Editor where the message begins. The second column of the array specifies the column in the Editor where the message ends. There is one row in the two-column array for each occurrence of a message.

If you specify multiple file names as input, or if you specify a cell array as input, `inform` contains a cell array of structures.

`msg=checkcode('filename','-string')` returns the information as a string to the variable `msg`. If you specify multiple file names as input, or if you specify a cell array as input, `msg` contains a string where each file's information is separated by 10 equal sign characters (=), a space, the file name, a space, and 10 equal sign characters.

If you omit the **`-struct`** or **`-string`** argument and you specify an output argument, the default behavior is **`-struct`**. If you omit the argument

and there are no output arguments, the default behavior is to display the information to the command line.

`[inform,filepaths]=checkcode('filename')` additionally returns `filepaths`, the absolute paths to the file names, in the same order as you specified them.

`inform=checkcode('filename','-id')` requests the message ID, where ID is a string of the form `ABC...`. When returned to a structure, the output also has the `id` field, which is the ID associated with the message.

`inform=checkcode('filename','-fullpath')` assumes that the input file names are absolute paths, so that `checkcode` does not try to locate them.

`inform=checkcode('filename','-notok')` runs `checkcode` for all lines in `filename`, even those lines that end with the `checkcode` suppression directive, `%#ok`.

`checkcode('filename','-cyc')` displays the McCabe complexity (also referred to as cyclomatic complexity) of each function in the file. Higher McCabe complexity values indicate higher complexity, and there is some evidence to suggest that programs with higher complexity values are more likely to contain errors. Frequently, you can lower the complexity of a function by dividing it into smaller, simpler functions. In general, smaller complexity values indicate programs that are easier to understand and modify. Some people advocate splitting up programs that have a complexity rating over 10.

`checkcode('filename','-codegen')` enables code generation messages for display in the Command Window.

`checkcode('filename','-eml')` `'-eml'` is not recommended. Use `'-codegen'` instead.

Examples

The following examples use `lengthofline.m`, which is a sample file with MATLAB code that can be improved. You can find it in *`matlabroot`*`/help/techdoc/matlab_env/examples`. If you want to run

the examples, save a copy of `lengthofline.m` to a location on your MATLAB path.

Running checkcode on a File with No Options

To run `checkcode` on the example file, `lengthofline.m`, run

```
checkcode('lengthofline')
```

MATLAB displays the Code Analyzer messages for `lengthofline.m` in the Command Window:

```
L 22 (C 1-9): The value assigned here to variable 'nothandle' might never be used.
L 23 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 24 (C 5-11): 'notline' might be growing inside a loop. Consider preallocating for speed.
L 24 (C 44-49): Use STRCMPI(str1,str2) instead of using LOWER in a call to STRCMP.
L 28 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 34 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 34 (C 24-31): Use dynamic fieldnames with structures instead of GETFIELD.
                Type 'doc struct' for more information.
L 38 (C 29): Use || instead of | as the OR operator in (scalar) conditional statements.
L 39 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 40 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 42 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 43 (C 13-15): 'dim' might be growing inside a loop. Consider preallocating for speed.
L 45 (C 13-15): 'dim' might be growing inside a loop.Consider preallocating for speed.
L 48 (C 52): There may be a parenthesis imbalance around here.
L 48 (C 53): There may be a parenthesis imbalance around here.
L 48 (C 54): There may be a parenthesis imbalance around here.
L 48 (C 55): There may be a parenthesis imbalance around here.
L 49 (C 17): Terminate statement with semicolon to suppress output (in functions).
L 49 (C 23): Use of brackets [] is unnecessary. Use parentheses to group, if needed.
```

For details about these messages and how to improve the code, see “Changing Code Based on Code Analyzer Messages” in the MATLAB Desktop Tools and Development Environment documentation.

Running checkcode with Options to Show IDs and Return Results to a Structure

To store the results to a structure and include message IDs, run

```
inform=checkcode('lengthofline', '-id')
```

MATLAB returns

```
inform =

19x1 struct array with fields:
    message
    line
    column
    id
```

To see values for the first message, run

```
inform(1)
```

MATLAB displays

```
ans =

    message: 'The value assigned here to variable 'nothandle' might never be used.'
       line: 22
     column: [1 9]
         id: 'NASGU'
```

Here, the message is for the value that appears on line 22 that extends from column 1–9 in the file.`NASGU` is the ID for the message `'The value assigned here to variable 'nothandle' might never be used.'`.

Displaying McCabe Complexity with checkcode

To display the McCabe complexity of a MATLAB code file, run `checkcode` with the `-cyc` option, as shown in the following example (assuming you have saved `lengthofline.m` to a local folder).

```
checkcode lengthofline.m -cyc
```

Results displayed in the Command Window show the McCabe complexity of the file, followed by the Code Analyzer messages, as shown here:

```
L 1 (C 23-34): The McCabe complexity of 'lengthofline' is 12.
L 22 (C 1-9): The value assigned here to variable 'nothandle' might never be used.
L 23 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 24 (C 5-11): 'notline' might be growing inside a loop. Consider preallocating for speed.
L 24 (C 44-49): Use STRCMPI(str1,str2) instead of using UPPER/LOWER in a call to STRCMP.
L 28 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 34 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 34 (C 24-31): Use dynamic fieldnames with structures instead of GETFIELD. Type 'doc struct' for mo
L 38 (C 29): Use || instead of | as the OR operator in (scalar) conditional statements.
L 39 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 40 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 42 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 43 (C 13-15): 'dim' might be growing inside a loop. Consider preallocating for speed.
L 45 (C 13-15): 'dim' might be growing inside a loop. Consider preallocating for speed.
L 48 (C 52): There may be a parenthesis imbalance around here.
L 48 (C 53): There may be a parenthesis imbalance around here.
L 48 (C 54): There may be a parenthesis imbalance around here.
L 48 (C 55): There may be a parenthesis imbalance around here.
L 49 (C 17): Terminate statement with semicolon to suppress output (in functions).
L 49 (C 23): Use of brackets [] is unnecessary.  Use parentheses to group, if needed.
```

See Also

`mlintrpt`, `profile`

How To

- For information on the suppression directive, `%#ok`, and suppressing messages from within your program, see "Adjust Code Analyzer Message Indicators and Messages".

checkin

Purpose

Check files into source control system (UNIX platforms)

Syntax

```
checkin('filename','comments','comment_text')
checkin({'filename1','filename2'},'comments','comment_text')
checkin('filename','comments', 'comment_text','option',
   'value')
```

Description

`checkin('filename','comments','comment_text')` checks in the file named `filename` to the source control system. Use the full path for `filename` and include the file extension. You must save the file before checking it in, but the file can be open or closed. The `comment_text` is a MATLAB string containing checkin comments for the source control system. You must supply **`comments`** and `comment_text`.

`checkin({'filename1','filename2'},'comments','comment_text')` checks in the files `filename1` through `filenamen` to the source control system. Use the full paths for the files and include file extensions. Comments apply to all files checked in.

`checkin('filename','comments', 'comment_text','option','value')` provides additional `checkin` options. For multiple file names, use an array of strings instead of `filename`, that is, `{'filename1','filename2',...}`. Options apply to all file names. The *`option`* and *`value`* arguments are shown in the following table.

option Argument	value Argument	Purpose
`'force'`	`'on'`	`filename` is checked in even if the file has not changed since it was checked out.
`'force'`	`'off'` (default)	`filename` is not checked in if there were no changes since checkout.

option Argument	value Argument	Purpose
'lock'	'on'	filename is checked in with comments, and is automatically checked out.
'lock'	'off' (default)	filename is checked in with comments but does not remain checked out.

Examples

Check In a File

Check the file /myserver/myfiles/clock.m into the source control system, with the comment Adjustment for leapyear:

```
checkin('/myserver/myfiles/clock.m','comments',...
'Adjustment for leapyear')
```

Check In Multiple Files

Check two files into the source control system, using the same comment for each:

```
checkin({'/myserver/myfiles/clock.m', ...
'/myserver/myfiles/calendar.m'},'comments',...
'Adjustment for leapyear')
```

Check In a File and Keep It Checked Out

Check the file /myserver/myfiles/clock.m into the source control system and keep the file checked out:

```
checkin('/myserver/myfiles/clock.m','comments',...
'Adjustment for leapyear','lock','on')
```

See Also

checkout | cmopts | undocheckout | verctrl

How To

- "Check In Files (UNIX Platforms)"

checkout

Purpose

Check files out of source control system (UNIX platforms)

Syntax

```
checkout('filename')
checkout({'filename1','filename2', ...})
checkout('filename','option','value',...)
```

Description

`checkout('filename')` checks out the file named `filename` from the source control system. Use the full path for `filename` and include the file extension. The file can be open or closed when you use `checkout`.

`checkout({'filename1','filename2', ...})` checks out the files named `filename1` through `filenamen` from the source control system. Use the full paths for the files and include the file extensions.

`checkout('filename','option','value',...)` provides additional `checkout` options. For multiple file names, use an array of strings instead of `filename`, that is, `{'filename1','filename2', ...}`. Options apply to all file names. The *`option`* and *`value`* arguments are shown in the following table.

option Argument	value Argument	Purpose
`'force'`	`'on'`	The checkout is forced, even if you already have the file checked out. This is effectively an `undocheckout` followed by a `checkout`.
`'force'`	`'off'` (default)	Prevents you from checking out the file if you already have it checked out.

option Argument	value Argument	Purpose
'lock'	'on' (default)	The checkout gets the file, allows you to write to it, and locks the file so that access to the file for others is read only.
'lock'	'off'	The checkout gets a read-only version of the file, allowing another user to check out the file for updating. You do not have to check the file in after checking it out with this option.
'revision'	'version_num'	Checks out the specified revision of the file.

If you end the MATLAB session, the file remains checked out. You can check in the file from within the MATLAB desktop during a later session, or directly from your source control system.

Examples

Check Out a File

Check out the file /myserver/myfiles/clock.m from the source control system:

```
checkout('/myserver/myfiles/clock.m')
```

Check Out Multiple Files

Check out /matlab/myfiles/clock.m and /matlab/myfiles/calendar.m from the source control system:

```
checkout({'/myserver/myfiles/clock.m',...
```

```
'/myserver/myfiles/calendar.m'})
```

Force a Checkout, Even If File Is Already Checked Out

Check out `/matlab/myfiles/clock.m` even if `clock.m` is already checked out to you:

```
checkout('/myserver/myfiles/clock.m','force','on')
```

Check Out Specified Revision of File

Check out revision `1.1` of `clock.m`:

```
checkout('/matlab/myfiles/clock.m','revision','1.1')
```

See Also

`checkin` | `cmopts` | `undocheckout` | `customverctrl` | `verctrl`

How To

- "Check Out Files (UNIX Platforms)"

Purpose

Cholesky factorization

Syntax

```
R = chol(A)
L = chol(A,'lower')
R = chol(A,'upper')
[R,p] = chol(A)
[L,p] = chol(A,'lower')
[R,p] = chol(A,'upper')
[R,p,S] = chol(A)
[R,p,s] = chol(A,'vector')
[L,p,s] = chol(A,'lower','vector')
[R,p,s] = chol(A,'upper','vector')
```

Description

`R = chol(A)` produces an upper triangular matrix `R` from the diagonal and upper triangle of matrix `A`, satisfying the equation `R'*R=A`. The `chol` function assumes that `A` is (complex Hermitian) symmetric. If it is not, `chol` uses the (complex conjugate) transpose of the upper triangle as the lower triangle. Matrix `A` must be positive definite.

`L = chol(A,'lower')` produces a lower triangular matrix `L` from the diagonal and lower triangle of matrix `A`, satisfying the equation `L*L'=A`. The `chol` function assumes that `A` is (complex Hermitian) symmetric. If it is not, `chol` uses the (complex conjugate) transpose of the lower triangle as the upper triangle. When `A` is sparse, this syntax of `chol` is typically faster. Matrix `A` must be positive definite. `R = chol(A,'upper')` is the same as `R = chol(A)`.

`[R,p] = chol(A)` for positive definite `A`, produces an upper triangular matrix `R` from the diagonal and upper triangle of matrix `A`, satisfying the equation `R'*R=A` and `p` is zero. If `A` is not positive definite, then `p` is a positive integer and MATLAB does not generate an error. When `A` is full, `R` is an upper triangular matrix of order `q=p-1` such that `R'*R=A(1:q,1:q)`. When `A` is sparse, `R` is an upper triangular matrix of size `q`-by-`n` so that the `L`-shaped region of the first `q` rows and first `q` columns of `R'*R` agree with those of `A`.

`[L,p] = chol(A,'lower')` for positive definite `A`, produces a lower triangular matrix `L` from the diagonal and lower triangle of matrix `A`,

satisfying the equation `L*L'=A` and `p` is zero. If `A` is not positive definite, then `p` is a positive integer and MATLAB does not generate an error. When `A` is full, `L` is a lower triangular matrix of order `q=p-1` such that `L*L'=A(1:q,1:q)`. When `A` is sparse, `L` is a lower triangular matrix of size `q`-by-`n` so that the `L`-shaped region of the first `q` rows and first `q` columns of `L*L'` agree with those of `A`. `[R,p] = chol(A,'upper')` is the same as `[R,p] = chol(A)`.

The following three-output syntaxes require sparse input `A`.

`[R,p,S] = chol(A)`, when `A` is sparse, returns a permutation matrix `S`. Note that the preordering `S` may differ from that obtained from `amd` since `chol` will slightly change the ordering for increased performance. When `p=0`, `R` is an upper triangular matrix such that `R'*R=S'*A*S`. When `p` is not zero, `R` is an upper triangular matrix of size `q`-by-`n` so that the `L`-shaped region of the first `q` rows and first `q` columns of `R'*R` agree with those of `S'*A*S`. The factor of `S'*A*S` tends to be sparser than the factor of `A`.

`[R,p,s] = chol(A,'vector')`, when `A` is sparse, returns the permutation information as a vector `s` such that `A(s,s)=R'*R`, when `p=0`. You can use the `'matrix'` option in place of `'vector'` to obtain the default behavior.

`[L,p,s] = chol(A,'lower','vector')`, when `A` is sparse, uses only the diagonal and the lower triangle of `A` and returns a lower triangular matrix `L` and a permutation vector `s` such that `A(s,s)=L*L'`, when `p=0`. As above, you can use the `'matrix'` option in place of `'vector'` to obtain a permutation matrix. `[R,p,s] = chol(A,'upper','vector')` is the same as `[R,p,s] = chol(A,'vector')`.

Note Using `chol` is preferable to using `eig` for determining positive definiteness.

Examples

Example 1

The `gallery` function provides several symmetric, positive, definite matrices.

```
A=gallery('moler',5)

A =

     1    -1    -1    -1    -1
    -1     2     0     0     0
    -1     0     3     1     1
    -1     0     1     4     2
    -1     0     1     2     5

C=chol(A)

ans =

     1    -1    -1    -1    -1
     0     1    -1    -1    -1
     0     0     1    -1    -1
     0     0     0     1    -1
     0     0     0     0     1
isequal(C'*C,A)

ans =

     1
```

For sparse input matrices, `chol` returns the Cholesky factor.

```
N = 100;
A = gallery('poisson', N);
```

`N` represents the number of grid points in one direction of a square `N`-by-`N` grid. Therefore, `A` is N^2 by N^2.

```
L = chol(A, 'lower');
D = norm(A - L*L', 'fro');
```

The value of D will vary somewhat among different versions of MATLAB but will be on order of 10^{-14}.

Example 2

The binomial coefficients arranged in a symmetric array create a positive definite matrix.

```
n = 5;
X = pascal(n)
X =
     1     1     1     1     1
     1     2     3     4     5
     1     3     6    10    15
     1     4    10    20    35
     1     5    15    35    70
```

This matrix is interesting because its Cholesky factor consists of the same coefficients, arranged in an upper triangular matrix.

```
R = chol(X)
R =
     1     1     1     1     1
     0     1     2     3     4
     0     0     1     3     6
     0     0     0     1     4
     0     0     0     0     1
```

Destroy the positive definiteness (and actually make the matrix singular) by subtracting 1 from the last element.

```
X(n,n) = X(n,n)-1

X =
     1     1     1     1     1
     1     2     3     4     5
```

```
     1     3     6    10    15
     1     4    10    20    35
     1     5    15    35    69
```

Now an attempt to find the Cholesky factorization of X fails.

```
chol(X)
Error using chol
Matrix must be positive definite.
```

See Also

cholinc | cholupdate

cholinc

Purpose

Sparse incomplete Cholesky and Cholesky-Infinity factorizations

---

Note `cholinc` has been removed.

- You can replace most instances of `cholinc` with `ichol`. Note that `cholinc` returned upper triangular factors, while `ichol` returns lower triangular factors by default.
- Remove any instances that use the syntax `cholinc(X,'inf')`. There is no replacement for this syntax.

Syntax

```
R = cholinc(X,droptol)
R = cholinc(X,options)
R = cholinc(X,'0')
[R,p] = cholinc(X,'0')
R = cholinc(X,'inf')
```

Description

`cholinc` produces two different kinds of incomplete Cholesky factorizations: the drop tolerance and the 0 level of fill-in factorizations. These factors may be useful as preconditioners for a symmetric positive definite system of linear equations being solved by an iterative method such as `pcg` (Preconditioned Conjugate Gradients). `cholinc` works only for sparse matrices.

`R = cholinc(X,droptol)` performs the incomplete Cholesky factorization of `X`, with drop tolerance `droptol`.

`R = cholinc(X,options)` allows additional options to the incomplete Cholesky factorization. `options` is a structure with up to three fields:

`droptol`	Drop tolerance of the incomplete factorization
`michol`	Modified incomplete Cholesky
`rdiag`	Replace zeros on the diagonal of `R`

Only the fields of interest need to be set.

`droptol` is a non-negative scalar used as the drop tolerance for the incomplete Cholesky factorization. This factorization is computed by performing the incomplete LU factorization with the pivot threshold option set to `0` (which forces diagonal pivoting) and then scaling the rows of the incomplete upper triangular factor, `U`, by the square root of the diagonal entries in that column. Since the nonzero entries `U(i,j)` are bounded below by `droptol*norm(X(:,j))` (see `luinc`), the nonzero entries `R(i,j)` are bounded below by the local drop tolerance `droptol*norm(X(:,j))/R(i,i)`.

Setting `droptol = 0` produces the complete Cholesky factorization, which is the default.

`michol` stands for modified incomplete Cholesky factorization. Its value is either `0` (unmodified, the default) or `1` (modified). This performs the modified incomplete LU factorization of `X` and scales the returned upper triangular factor as described above.

`rdiag` is either `0` or `1`. If it is `1`, any zero diagonal entries of the upper triangular factor R are replaced by the square root of the local drop tolerance in an attempt to avoid a singular factor. The default is `0`.

`R = cholinc(X,'0')` produces the incomplete Cholesky factor of a real sparse matrix that is symmetric and positive definite using no fill-in. The upper triangular `R` has the same sparsity pattern as `triu(X)`, although `R` may be zero in some positions where `X` is nonzero due to cancellation. The lower triangle of `X` is assumed to be the transpose of the upper. Note that the positive definiteness of `X` does not guarantee the existence of a factor with the required sparsity. An error message results if the factorization is not possible. If the factorization is successful, `R'*R` agrees with `X` over its sparsity pattern.

`[R,p] = cholinc(X,'0')` with two output arguments, never produces an error message. If `R` exists, `p` is `0`. If `R` does not exist, then `p` is a positive integer and `R` is an upper triangular matrix of size `q`-by-`n` where `q = p-1`. In this latter case, the sparsity pattern of `R` is that of the `q`-by-`n` upper triangle of `X`. `R'*R` agrees with `X` over the sparsity pattern of its first `q` rows and first `q` columns.

`R = cholinc(X,'inf')` produces the Cholesky-Infinity factorization. This factorization is based on the Cholesky factorization, and additionally handles real positive semi-definite matrices. It may be useful for finding a solution to systems which arise in interior-point methods. When a zero pivot is encountered in the ordinary Cholesky factorization, the diagonal of the Cholesky-Infinity factor is set to `Inf` and the rest of that row is set to `0`. This forces a `0` in the corresponding entry of the solution vector in the associated system of linear equations. In practice, `X` is assumed to be positive semi-definite so even negative pivots are replaced with a value of `Inf`.

Tips

The incomplete factorizations may be useful as preconditioners for solving large sparse systems of linear equations. A single `0` on the diagonal of the upper triangular factor makes it singular. The incomplete factorization with a drop tolerance prints a warning message if the upper triangular factor has zeros on the diagonal. Similarly, using the `rdiag` option to replace a zero diagonal only gets rid of the symptoms of the problem, but it does not solve it. The preconditioner may not be singular, but it probably is not useful, and a warning message is printed.

The Cholesky-Infinity factorization is meant to be used within interior-point methods. Otherwise, its use is not recommended.

Examples

Example 1

Start with a symmetric positive definite matrix, `S`.

```
S = delsq(numgrid('C',15));
```

S is the two-dimensional, five-point discrete negative Lapacian on the grid generated by numgrid('C',15).

Compute the Cholesky factorization and the incomplete Cholesky factorization of level 0 to compare the fill-in. Make `S` singular by zeroing out a diagonal entry and compute the (partial) incomplete Cholesky factorization of level `0`.

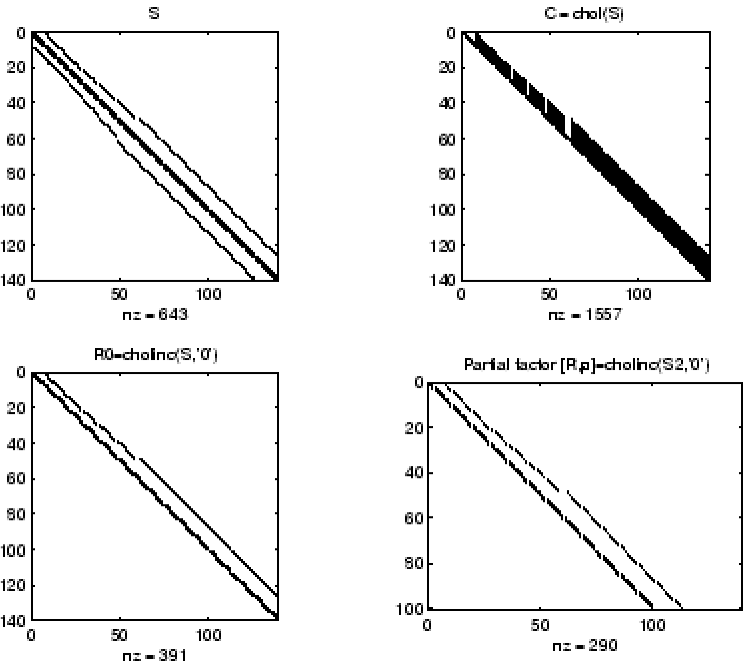
```
C = chol(S);
R0 = cholinc(S,'0');
```

```
S2 = S; S2(101,101) = 0;
[R,p] = cholinc(S2,'0');
```

Fill-in occurs within the bands of `S` in the complete Cholesky factor, but none in the incomplete Cholesky factor. The incomplete factorization of the singular `S2` stopped at row `p = 101` resulting in a 100-by-139 partial factor.

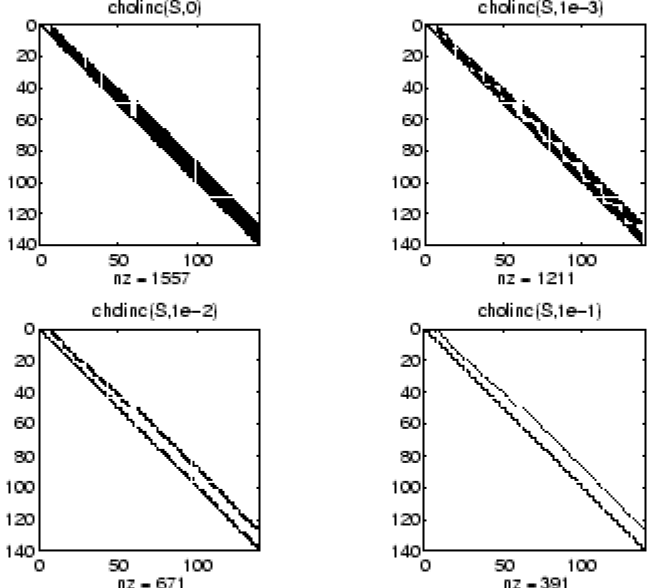
```
D1 = (R0'*R0).*spones(S)-S;
D2 = (R'*R).*spones(S2)-S2;
```

`D1` has elements of the order of `eps`, showing that `R0'*R0` agrees with `S` over its sparsity pattern. `D2` has elements of the order of `eps` over its first 100 rows and first 100 columns, `D2(1:100,:)` and `D2(:,1:100)`.

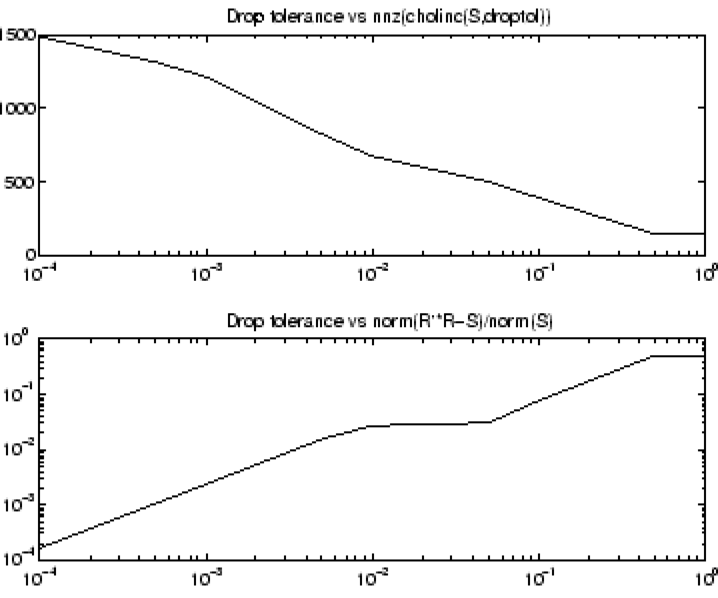


Example 2

The first subplot below shows that `cholinc(S,0)`, the incomplete Cholesky factor with a drop tolerance of `0`, is the same as the Cholesky factor of `S`. Increasing the drop tolerance increases the sparsity of the incomplete factors, as seen below.



Unfortunately, the sparser factors are poor approximations, as is seen by the plot of drop tolerance versus `norm(R'*R-S,1)/norm(S,1)` in the next figure.



Example 3

The Hilbert matrices have (i,j) entries 1/(i+j-1) and are theoretically positive definite:

```
H3 = hilb(3)
H3 =
    1.0000    0.5000    0.3333
    0.5000    0.3333    0.2500
    0.3333    0.2500    0.2000
R3 = chol(H3)
R3 =
    1.0000    0.5000    0.3333
         0    0.2887    0.2887
         0         0    0.0745
```

In practice, the Cholesky factorization breaks down for larger matrices:

```
H20 = sparse(hilb(20));
```

```
[R,p] = chol(H20);
p =
    14
```

For `hilb(20)`, the Cholesky factorization failed in the computation of row 14 because of a numerically zero pivot. You can use the Cholesky-Infinity factorization to avoid this error. When a zero pivot is encountered, `cholinc` places an `Inf` on the main diagonal, zeros out the rest of the row, and continues with the computation:

```
Rinf = cholinc(H20,'inf');
```

In this case, all subsequent pivots are also too small, so the remainder of the upper triangular factor is:

```
full(Rinf(14:end,14:end))
ans =
   Inf     0     0     0     0     0     0
     0   Inf     0     0     0     0     0
     0     0   Inf     0     0     0     0
     0     0     0   Inf     0     0     0
     0     0     0     0   Inf     0     0
     0     0     0     0     0   Inf     0
     0     0     0     0     0     0   Inf
```

Limitations

`cholinc` works on square sparse matrices only. For `cholinc(X,'0')` and `cholinc(X,'inf')`, `X` must be real.

Algorithms

`R = cholinc(X,droptol)` is obtained from `[L,U] = luinc(X,options)`, where `options.droptol = droptol` and `options.thresh = 0`. The rows of the uppertriangular `U` are scaled by the square root of the diagonal in that row, and this scaled factor becomes `R`.

`R = cholinc(X,options)` is produced in a similar manner, except the `rdiag` option translates into the `udiag` option and the `milu` option takes the value of the `michol` option.

`R = cholinc(X,'0')` is based on the "KJI" variant of the Cholesky factorization. Updates are made only to positions which are nonzero in the upper triangle of `X`.

`R = cholinc(X,'inf')` is based on the algorithm in Zhang [2].

References

[1] Saad, Yousef, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996. Chapter 10, "Preconditioning Techniques"

[2] Zhang, Yin, *Solving Large-Scale Linear Programs by Interior-Point Methods Under the MATLAB Environment*, Department of Mathematics and Statistics, University of Maryland Baltimore County, Technical Report TR96-01

See Also

`chol` | `ilu` | `luinc` | `pcg`

cholupdate

Purpose

Rank 1 update to Cholesky factorization

Syntax

```
R1 = cholupdate(R,x)
R1 = cholupdate(R,x,'+')
R1 = cholupdate(R,x,'-')
[R1,p] = cholupdate(R,x,'-')
```

Description

`R1 = cholupdate(R,x)` where `R = chol(A)` is the original Cholesky factorization of `A`, returns the upper triangular Cholesky factor of `A + x*x'`, where `x` is a column vector of appropriate length. `cholupdate` uses only the diagonal and upper triangle of `R`. The lower triangle of `R` is ignored.

`R1 = cholupdate(R,x,'+')` is the same as `R1 = cholupdate(R,x)`.

`R1 = cholupdate(R,x,'-')` returns the Cholesky factor of `A - x*x'`. An error message reports when R is not a valid Cholesky factor or when the downdated matrix is not positive definite and so does not have a Cholesky factorization.

`[R1,p] = cholupdate(R,x,'-')` will not return an error message. If `p` is `0`, `R1` is the Cholesky factor of `A - x*x'`. If `p` is greater than `0`, `R1` is the Cholesky factor of the original `A`. If `p` is `1`, `cholupdate` failed because the downdated matrix is not positive definite. If `p` is `2`, `cholupdate` failed because the upper triangle of `R` was not a valid Cholesky factor.

Tips

`cholupdate` works only for full matrices.

Examples

```
A = pascal(4)
A =

     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20

R = chol(A)
R =
```

```
     1     1     1     1
     0     1     2     3
     0     0     1     3
     0     0     0     1
x = [0 0 0 1]';
```

This is called a rank one update to `A` since `rank(x*x')` is `1`:

```
A + x*x'
ans =



     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    21
```

Instead of computing the Cholesky factor with `R1 = chol(A + x*x')`, we can use `cholupdate`:

```
R1 = cholupdate(R,x)
R1 =

    1.0000    1.0000    1.0000    1.0000
         0    1.0000    2.0000    3.0000
         0         0    1.0000    3.0000
         0         0         0    1.4142
```

Next destroy the positive definiteness (and actually make the matrix singular) by subtracting `1` from the last element of `A`. The downdated matrix is:

```
A - x*x'
ans =

     1     1     1     1
     1     2     3     4
```

```
     1     3     6    10
     1     4    10    19
```

Compare `chol` with `cholupdate`:

```
R1 = chol(A-x*x')
Error using chol
Matrix must be positive definite.
R1 = cholupdate(R,x,'-')
Error using cholupdate
Downdated matrix must be positive definite.
```

However, subtracting `0.5` from the last element of `A` produces a positive definite matrix, and we can use `cholupdate` to compute its Cholesky factor:

```
x = [0 0 0 1/sqrt(2)]';
R1 = cholupdate(R,x,'-')
R1 =
    1.0000    1.0000    1.0000    1.0000
         0    1.0000    2.0000    3.0000
         0         0    1.0000    3.0000
         0         0         0    0.7071
```

Algorithms

`cholupdate` uses the algorithms from the LINPACK subroutines `ZCHUD` and `ZCHDD`. `cholupdate` is useful since computing the new Cholesky factor from scratch is an $O(N^3)$ algorithm, while simply updating the existing factor in this way is an $O(N^2)$ algorithm.

References

[1] Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

See Also

`chol` | `qrupdate`

Purpose

Shift array circularly

Syntax

```
B = circshift(A,shiftsize)
```

Description

`B = circshift(A,shiftsize)` circularly shifts the values in the array, `A`, by `shiftsize` elements. `shiftsize` is a vector of integer scalars where the `n`-th element specifies the shift amount for the `n`-th dimension of array `A`. If an element in `shiftsize` is positive, the values of `A` are shifted down (or to the right). If it is negative, the values of `A` are shifted up (or to the left). If it is `0`, the values in that dimension are not shifted.

Examples

Circularly shift first dimension values down by 1.

```
A = [ 1 2 3;4 5 6; 7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9

B = circshift(A,1)
B =
     7     8     9
     1     2     3
     4     5     6
```

Circularly shift first dimension values down by 1 and second dimension values to the left by 1.

```
B = circshift(A,[1 -1]);
B =
    8     9     7
    2     3     1
    5     6     4
```

See Also

`fftshift` | `shiftdim` | `permute` | `reshape`

TriRep.circumcenters

Purpose

(Will be removed) Circumcenters of specified simplices

Note `circumcenters(TriRep)` will be removed in a future release. Use `circumcenter(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
CC = circumcenters(TR, SI)
[CC RCC] = circumcenters(TR, SI)
```

Description

`CC = circumcenters(TR, SI)` returns the coordinates of the circumcenter of each specified simplex `SI`. `CC` is an `m`-by-`n` matrix, where `m` is of length `length(SI)`, the number of specified simplices, and `n` is the dimension of the space where the triangulation resides.

`[CC RCC] = circumcenters(TR, SI)` returns the circumcenters and the corresponding radii of the circumscribed circles or spheres.

Input Arguments

`TR`	Triangulation object.
`SI`	Column vector of simplex indices that index into the triangulation matrix `TR.Triangulation`. If `SI` is not specified the circumcenter information for the entire triangulation is returned, where the circumcenter associated with simplex `i` is the `i`'th row of `CC`.

Output Arguments

`CC`	`m`-by-`n` matrix. `m` is the number of specified simplices and `n` is the dimension of the space where the triangulation resides. Each row `CC(i,:)` represents the coordinates of the circumcenter of simplex `SI(i)`.
`RCC`	Vector of length `length(SI)`, the number of specified simplices containing radii of the circumscribed circles or spheres.

Definitions

A simplex is a triangle/tetrahedron or higher-dimensional equivalent.

Examples

Example 1

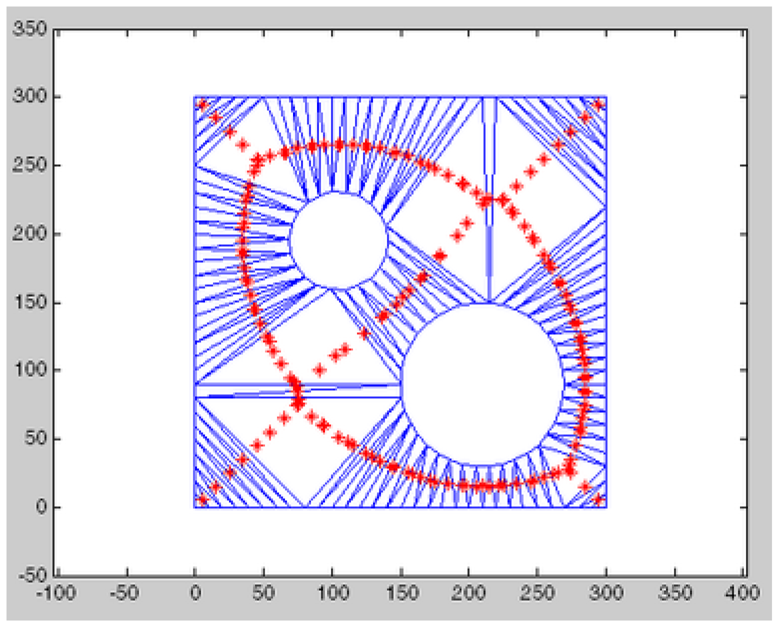
Load a 2-D triangulation.

```
load trimesh2d
trep = TriRep(tri, x,y)
```

Compute the circumcenters.

```
cc = circumcenters(trep);
triplot(trep);
axis([-50 350 -50 350]);
axis equal;
hold on;
plot(cc(:,1),cc(:,2),'*r');
hold off;
```

The circumcenters represent points on the medial axis of the polygon.



Example 2

Query a 3-D triangulation created with `DelaunayTri`. Compute the circumcenters of the first five tetrahedra.

```
X = rand(10,3);
dt = DelaunayTri(X);
cc = circumcenters(dt, [1:5]')
```

See Also

`incenter` | `delaunayTriangulation` | `triangulation`

Purpose

Clear current axes

Syntax

```
cla
cla reset
cla(ax)
cla(ax,'reset')
```

Description

`cla` deletes from the current axes all graphics objects whose handles are not hidden (i.e., their `HandleVisibility` property is set to `on`).

`cla reset` deletes from the current axes all graphics objects regardless of the setting of their `HandleVisibility` property and resets all axes properties, except `Position` and `Units`, to their default values.

`cla(ax)` or `cla(ax,'reset')` clears the single axes with handle `ax`.

Tips

The `cla` command behaves the same way when issued on the command line as it does in callback routines — it does not recognize the `HandleVisibility` setting of `callback`. This means that when issued from within a callback routine, `cla` deletes only those objects whose `HandleVisibility` property is set to `on`.

Alternatives

Remove axes and clear objects from them in *plot edit* mode. For details, see "Working in Plot Edit Mode".

See Also

`clf` | `hold` | `newplot` | `reset`

clabel

Purpose

Contour plot elevation labels

Syntax

```
clabel(C)
clabel(C,h)
clabel(C,v)
clabel(C,h,v)
clabel( ___ ,Name,Value)

clabel(C,'manual')
clabel(C,h,'manual')

text_handles = clabel( ___ )
```

Description

`clabel(C)` labels all contours displayed in the current contour plot. Labels are upright and displayed with `'+'` symbols. `clabel` randomly selects label positions.

`clabel(C,h)` rotates the labels and inserts them in the contour lines. This syntax inserts only those labels that fit within the contour, depending on the size of the contour.

`clabel(C,v)` labels only the contour levels specified by the vector, `v`.

`clabel(C,h,v)` labels only the contour levels specified by vector `v`, rotates the labels, and inserts them in the contour lines.

`clabel( ___ ,Name,Value)` specifies the text object properties and the `'LabelSpacing'` contourgroup property, using one or more `Name,Value` pair arguments, in addition to any of the input arguments in previous syntaxes.

`clabel(C,'manual')` places contour labels at locations you select with a mouse. Click the mouse or press the space bar to label the contour

closest to the center of the crosshair. Press the **Return** key while the cursor is within the figure window to terminate labeling.

`clabel(C,h,'manual')` places contour labels at locations you select with a mouse. Press the **Return** key while the cursor is within the figure window to terminate labeling. The labels are rotated and inserted in the contour lines.

`text_handles = clabel(___)` additionally returns an array containing the handles of the text objects created, using any of the input arguments in the previous syntaxes. If you call `clabel` without the `h` argument, `text_handles` also contains the handles of line objects used to create the `'+'` symbols.

Input Arguments

C - Contour matrix

2-by-`n` matrix

Contour matrix containing the data that defines the contour lines. `C` is returned by the `contour`, `contour3`, or `contourf` function.

h - Handle to the contourgroup object

Handle to the contourgroup object returned by the `contour`, `contour3`, or `contourf` function.

v - Contour level values

vector

Contour level values, specified as a row or column vector of individual values.

Example: `[0,10,20]`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can

specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Color','red','Rotation',45` adds red labels to a contour plot, where each label is rotated 45 degrees.

In addition to the following, you can specify other text object properties using `Name,Value` pair arguments. See Text Properties.

'Color' - Text color

`[0 0 0]` (black) (default) | 3-element RGB vector | string

Text color, specified as the comma-separated pair consisting of `'Color'` and a 3-element RGB vector or a string containing the short or long name of the color. The RGB vector is a 3-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0 1]`.

The following table lists the predefined colors and their RGB equivalents.

RGB Value	Short Name	Long Name
`[1 1 0]`	`y`	`yellow`
`[1 0 1]`	`m`	`magenta`
`[0 1 1]`	`c`	`cyan`
`[1 0 0]`	`r`	`red`
`[0 1 0]`	`g`	`green`
`[0 0 1]`	`b`	`blue`
`[1 1 1]`	`w`	`white`
`[0 0 0]`	`k`	`black`

Example: `'Color',[0 1 0]`

Example: `'Color','green'`

'FontName' - Font name

'Helvetica' (default) | string | 'FixedWidth'

Font name, specified as the comma-separated pair consisting of 'FontName' and a string. The string specifies the name of the font to use for the text object. To display and print properly, this must be a font that your system supports.

To use a fixed-width font that looks good in any locale, use the case-sensitive string 'FixedWidth'. This eliminates the need to hard-code the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding.

Example: 'FontName','Courier'

'FontSize' - Font size

10 points (default) | scalar

Font size, specified as the comma-separated pair consisting of 'FontSize' and a scalar in units determined by the FontUnits property. The default value for FontUnits is points.

Example: 'FontSize',12.5

'FontUnits' - Font size units

'points' (default) | 'normalized' | 'inches' | 'centimeters' | 'pixels'

Font size units, specified as the comma-separated pair consisting of 'FontUnits' and one of the following strings:

- 'points'
- 'normalized'
- 'inches'
- 'centimeters'
- 'pixels'

When the value of FontUnits is 'normalized', MATLAB interprets the value of FontSize as a fraction of the height of the parent axes. When you resize the axes, MATLAB modifies the screen FontSize

accordingly. `points`, `inches`, `centimeters`, and `pixels` are absolute units. 1 point = $^1/_{72}$ inch

---

Note When setting both the `FontSize` and the `FontUnits`, you must set the `FontUnits` property first so that MATLAB can correctly interpret the specified `FontSize`. For example, to set the font size to 0.3 inches, call `'FontUnits','inches','FontSize',0.3` in the argument list.

'FontWeight' - Weight of text characters

`'normal'` (default) | `'bold'` | `'light'` | `'demi'`

Weight of text characters, specified as the comma-separated pair consisting of `'FontWeight'` and one of the following strings:

- `'normal'`
- `'bold'`
- `'light'`
- `'demi'`

MATLAB uses the `FontWeight` property to select a font from those available on your particular system. Generally, setting this property to `'bold'` or `'demi'` causes MATLAB to use a bold font.

Example: `'FontWeight','bold'`

'LabelSpacing' - Spacing between labels

144 (default) | scalar

Spacing between labels on each contour line, specified as the comma-separated pair consisting of `'LabelSpacing'` and a scalar. Specify the label spacing in points, where 1 point = $^1/_{72}$ inch.

Example: `'LabelSpacing',72`

'Rotation' - Text orientation

0 (default) | scalar

Text orientation, specified as the comma-separated pair consisting of 'Rotation' and a scalar. Specify values of rotation in degrees. Positive values result in counterclockwise rotation.

Example: 'Rotation',45

Output Arguments

text_handles - Handles of text objects

Handles of the text objects that clabel creates. The UserData properties of the text objects contain the contour values displayed.

If you call clabel without the h argument, text_handles also contains the handles of line objects used to create the '+' symbols.

Examples

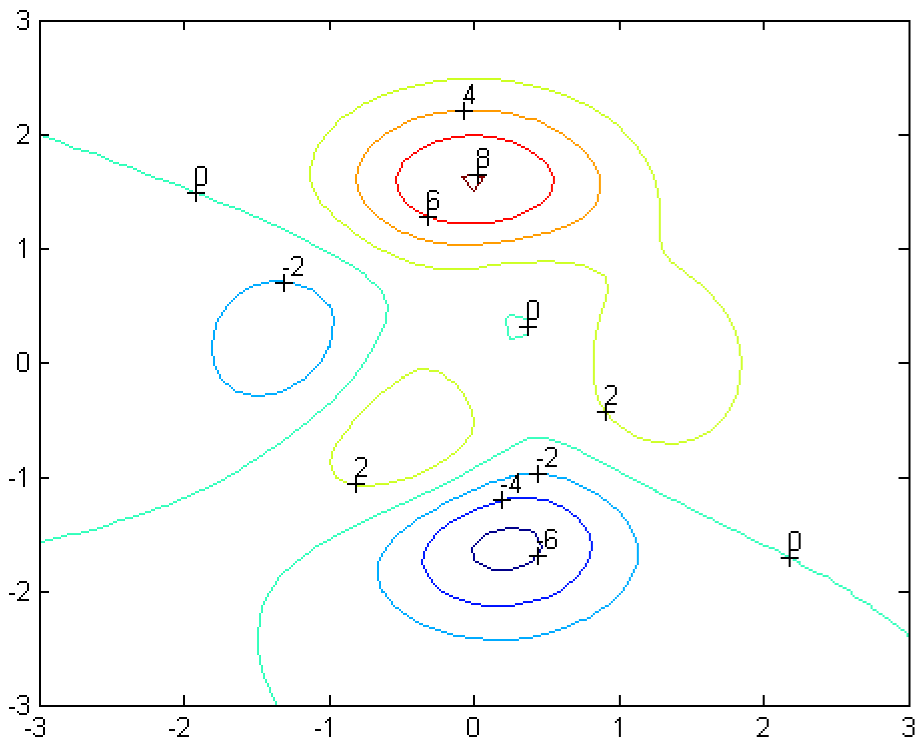
Label Contour Plot with Vertical Text and '+' Symbols

Generate a contour plot and obtain the contour matrix, C.

```
[x,y,z] = peaks;
figure
C = contour(x,y,z);
```

Label the contour plot.

```
clabel(C)
```



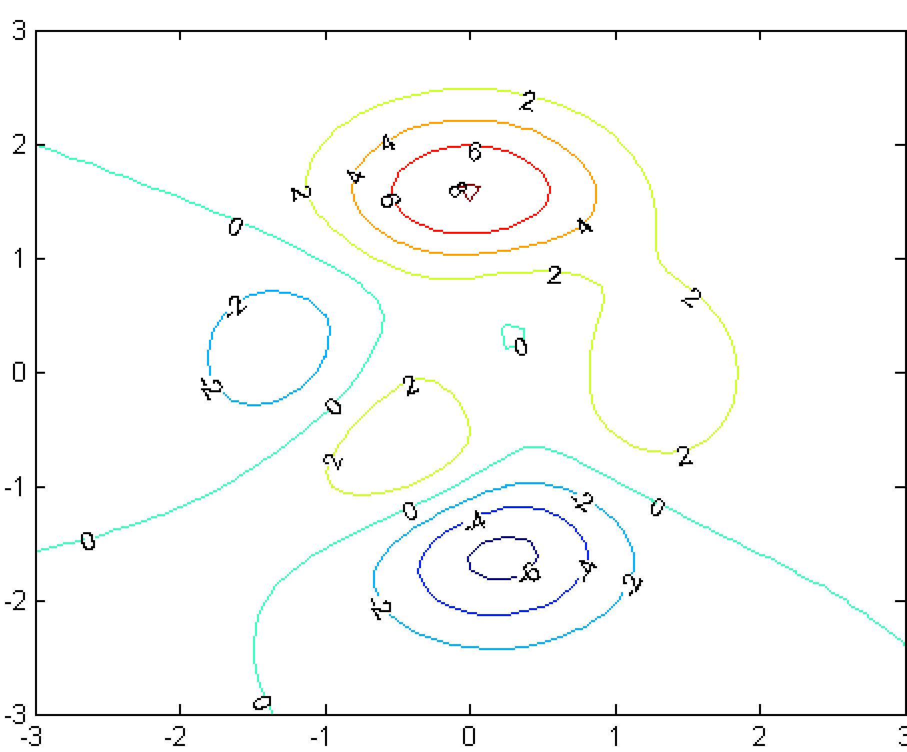
Label Contour Plot with Rotated Text Inserted in Contour Lines

Generate a contour plot and obtain the contour matrix, C, and the handle to the contourgroup object, h.

```
[x,y,z] = peaks;
figure
[C,h] = contour(x,y,z);
```

Label the contour plot.

```
clabel(C,h);
```

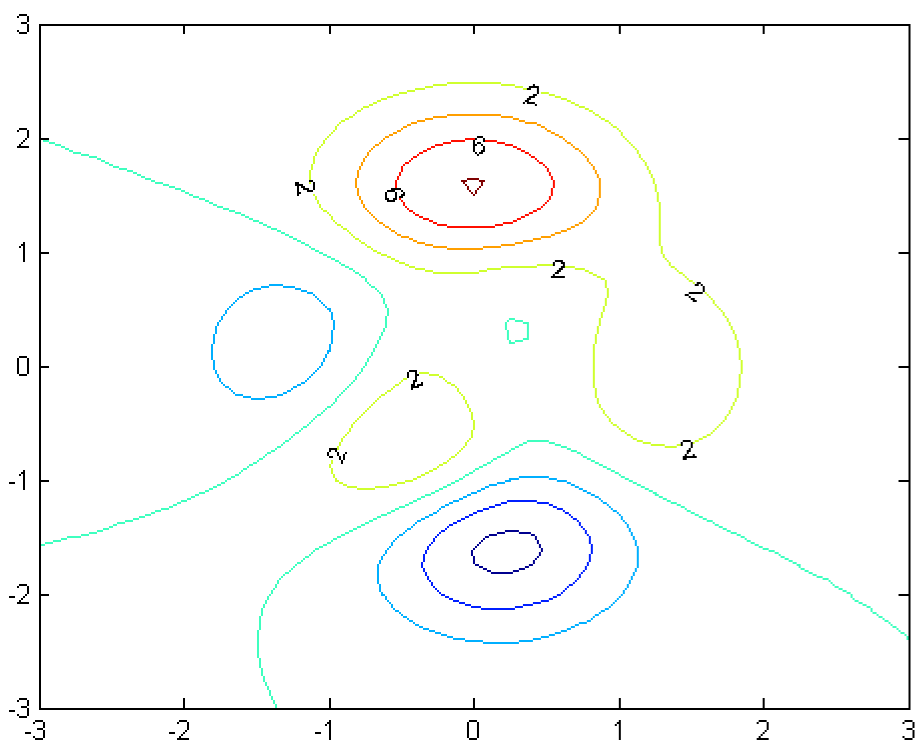


Label Specific Contour Levels

Label only the contours with contour levels 2 or 6.

```
[x,y,z] = peaks;
figure
[C,h] = contour(x,y,z);
```

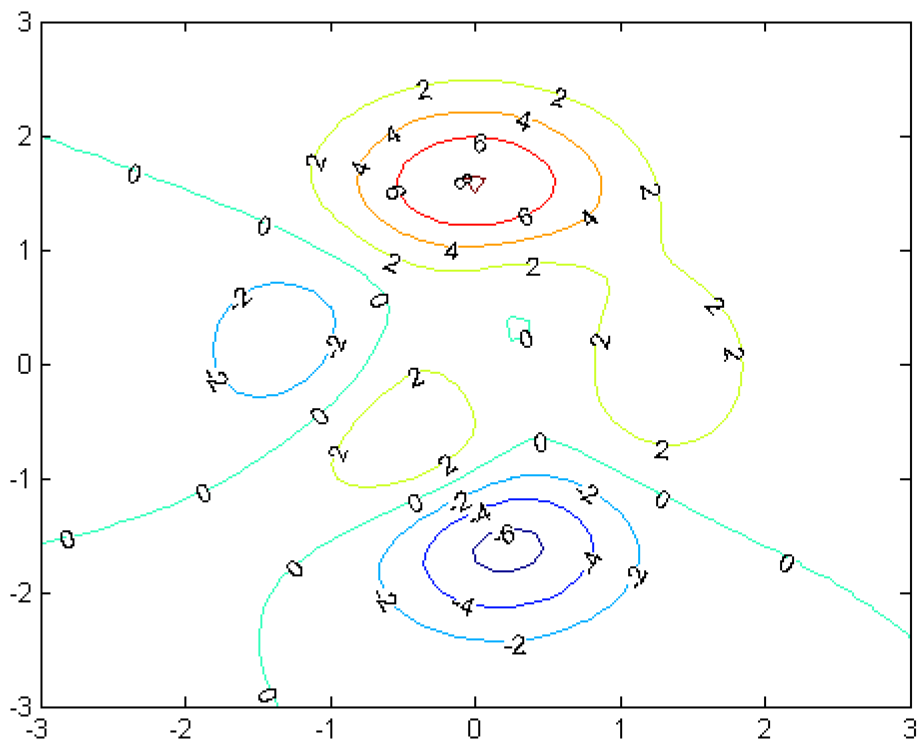
```
v = [2,6];
clabel(C,h,v)
```



Set Label Spacing

Set the label spacing to 72 points (1 inch).

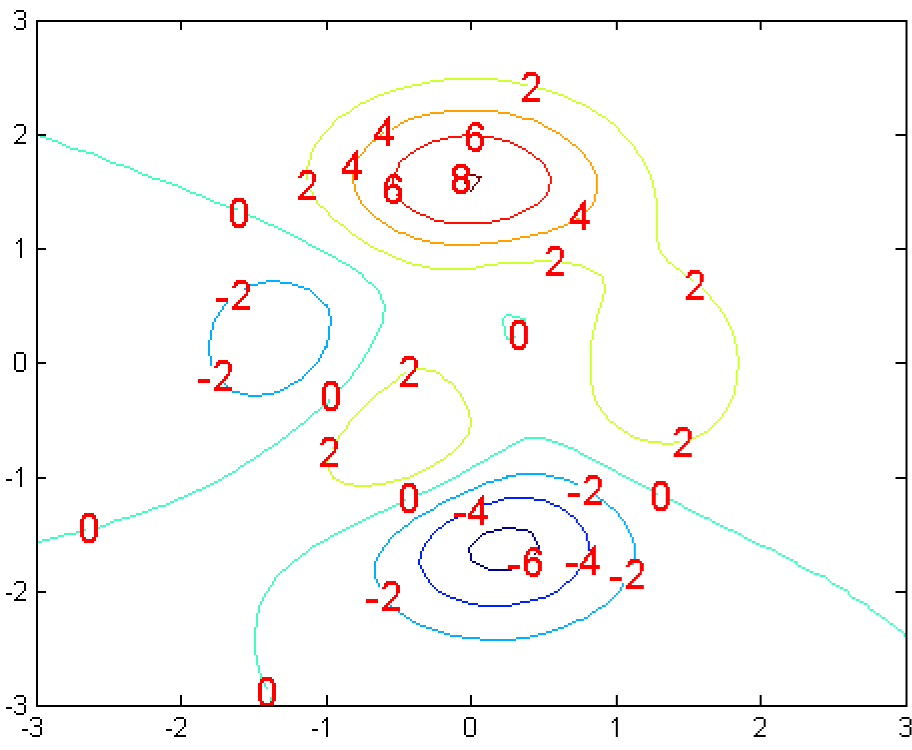
```
[x,y,z] = peaks;
figure
[C,h] = contour(x,y,z);
clabel(C,h,'LabelSpacing',72)
```



Label Contour Plot and Set Text Properties

Use `Name,Value` arguments to set the font size, font color, and text orientation of the labels.

```
[x,y,z] = peaks;
figure
[C,h] = contour(x,y,z);
clabel(C,h,'FontSize',15,'Color','r','Rotation',0)
```



`'FontSize',15,'Color','r'` adds 15-point red labels to the contour plot. `'Rotation',0` makes the text upright.

Label Contour Plot and Return Object Handles

Label a contour plot and return the handles of the objects created.

```
[x,y,z] = peaks;
figure
[C,h] = contour(x,y,z);
text_handles = clabel(C,h);
```

MATLAB returns an array containing the handles of the text objects created.

Set the color of the labels to blue, using the handle to the text objects.

```
set(text_handles,'Color','blue')
```

See Also

`contour` | `contourc` | `contourf` | `contour3`

Related Examples

- "Drawing Text in a Box"

Concepts

Text Properties

class

Purpose

Determine class of object

Syntax

```
ClassName = class(object)
obj = class(s,'class_name')
obj = class(s,'class_name',parent1,parent2,...)
obj = class(struct([]),'class_name',parent1,parent2,...)
obj_struct = class(struct_array,'class_name',parent_array)
```

Description

`ClassName = class(object)` returns a string specifying the class of `object`. See "Fundamental MATLAB Classes" for more information on MATLAB classes.

Note Before MATLAB 7.6 (classes defined without a `classdef` statement), class constructors called the `class` function to create the object. The following `class` function syntaxes apply only within classes defined before Version 7.6.

`obj = class(s,'class_name')` creates an array of class *`class_name`* objects using the `struct` `s` as a pattern to determine the size of `obj`.

`obj = class(s,'class_name',parent1,parent2,...)` inherits the methods and fields of the parent objects `parent1`, `parent2`, and so on. The size of the parent objects must match the size of `s` or be a scalar (1–by-1), in which case, MATLAB performs scalar expansion.

`obj = class(struct([]),'class_name',parent1,parent2,...)` constructs object containing only fields that it inherits from the parent objects. All parents must have the same, nonzero size, which determines the size of the returned object `obj`.

`obj_struct = class(struct_array,'class_name',parent_array)` maps every element of the `parent_array` to a corresponding element in the `struct_array` to produce the output array of objects, `obj_struct`.

All arrays must be of the same size. If either the `struct_array` or the `parent_array` is of size 1–by–1, then MATLAB performs scalar expansion to match the array sizes.

To create an object array of size 0–by-0, set the size of the `struct_array` and `parent_array` to 0–by-0.

Examples

Return the class of Java object `obj`:

```
import java.lang.*;
obj = String('mystring');
class(obj)

ans =

java.lang.String
```

Return class of any MATLAB variable:

```
h = @sin;
class(h)

ans =

function_handle
```

See Also

`isa` | `isobject` | `metaclass`

- "Class Syntax Fundamentals"

classdef

Purpose

Class definition keywords

Syntax

```
classdef classname
   properties
      PropName
   end
   methods
      methodName
   end
   events
      EventName
   end
   enumeration
      EnumName (arg)
   end
end
```

Description

`classdef` *`classname`* begins the class definition and an `end` keyword terminates the `classdef` block. Only blank lines and comments can precede `classdef`. Enter a class definition in a file having the same name as the class, with a filename extension of `.m`. Class definition files can be in folders on the MATLAB path or in @ folders whose parent folder is on the MATLAB path. See "Class Files" for more information. See "Classdef Block" and "Class Definition" for more information on classes.

`properties` begins a property definition block, an `end` keyword terminates the `properties` block. Class definitions can contain multiple property definition blocks, each specifying different attribute settings that apply to the properties in that particular block. See "Defining Properties" for more information.

`methods` begins a methods definition block, an `end` keyword terminates the `methods` block. This block contains functions that implement class methods. Class definitions can contain multiple method blocks, each specifying different attribute settings that apply to the methods in that particular block. It is possible to define method functions in separate files. See "How to Use Methods" for more information.

events begins an events definition block, an end keyword terminates the events block. This block contains event names defined by the class. Class definitions can contain multiple event blocks, each specifying different attribute settings that apply to the events in that particular block. See "Events and Listeners — Syntax and Techniques" for more information.

enumeration begins an enumeration definition block, an end keyword terminates the enumeration block. See "Enumerations" for more information.

properties, methods, events, and enumeration are also the names of MATLAB functions used to query the respective class members for a given object or class name.

To see the attributes of all class components in a popup window, click this link: Attribute Tables

Examples

Use these keywords to define classes.

```
classdef (Attributes) class_name
   properties (Attributes)
      PropertyName
   end
   methods (Attributes)
      function obj = methodName(obj,arg2,...)
         ...
      end
   end
   events (Attributes)
      EventName
   end
   enumeration
      EnumName
   end
end
```

See Also

properties | methods | events

Tutorials

- “Class Syntax Fundamentals”
-

clc

Purpose

Clear Command Window

Syntax

```
clc
```

Description

`clc` clears all input and output from the Command Window display, giving you a "clean screen."

After using `clc`, you cannot use the scroll bar to see the history of functions, but you still can use the up arrow to recall statements from the command history.

Examples

Use `clc` in a MATLAB code file to always display output in the same starting position on the screen.

See Also

`clear` | `clf` | `close` | `home`

clear

Purpose

Remove items from workspace, freeing up system memory

Syntax

```
clear
clear name1 ... nameN
clear -regexp expr1 ... exprN
clear ItemType
```

Description

`clear` removes all variables from the current workspace, releasing them from system memory. When called from within a function with persistent variables, `clear` reinitializes the persistent variables.

`clear name1 ...  nameN` removes the variables, scripts, functions, or MEX-functions `name1 ...  nameN` from memory. `clear` removes debugging breakpoints in MATLAB files.

- If function `name` is locked by `mlock`, then it remains in memory.
- If variable `name` is global, then `clear` removes it from the current workspace, but it remains in the global workspace.

`clear -regexp expr1 ...  exprN` clears all variables that match any of the regular expressions listed. This option only clears variables.

`clear ItemType` clears the types of items indicated by `ItemType`, such as `all`, `functions`, or `classes`.

Input Arguments

name1 ... nameN - Names of variables, scripts, functions, or MEX-functions to clear

string

Names of variables, scripts, functions, or MEX-functions to clear, specified as strings.

Use a partial path to distinguish between different overloaded versions of a function. For example, `clear polynom/display` clears only the `display` method for `polynom` objects, leaving any other implementations in memory.

expr1 ... exprN - Regular expressions matching names of variables to clear

string

Regular expressions matching names of variables to clear, specified as strings.

ItemType - Type of items to clear

`all` | `classes` | `functions` | `global` | `import` | `java` | `mex` | `variables`

Type of items to clear, specified as one of the following strings.

ItemType	Items Cleared
`all`	Removes all variables from the base workspace, and all compiled scripts, functions, and MEX-functions from memory. If called from a function, `clear all` clears the function workspace instead of the base workspace. `clear all` removes debugging breakpoints in MATLAB files and reinitializes persistent variables. When issued from the Command Window prompt, `clear all` also removes the Java packages import list.
`classes`	The same as `clear all`, but also clears MATLAB class definitions. `clear classes` issues a warning and does not remove class definitions for objects outside the workspace (for example, in user data or persistent variables in a locked file). Call `clear classes` whenever you change a class definition, including when the number or names of properties, methods, or events or any of their attributes change.

ItemType	Items Cleared
`functions`	Clears all the currently compiled functions, scripts, and MEX-functions from memory. If called from a function, `clear functions` removes all functions from memory. It also removes debugging breakpoints in the code file and reinitializes persistent variables.
`global`	Removes all global variables in the base and global workspaces. If called from a function, `clear global` also removes all global variables in the function workspace. Use `clear global  regexp expr1 ...  exprN` to clear all variables that match any of the regular expressions listed.
`import`	Removes the packages import list. Use this keyword only from the command prompt. Using this keyword in a function returns an error.
`java`	The same as `clear all`, but also clears the definitions of all Java classes defined by files on the Java dynamic class path. If any Java objects exist outside the workspace (for example, in user data or persistent variables in a locked code file), `clear` issues a warning and does not remove the Java class definition. Issue a `clear java` command after modifying any files on the Java dynamic class path.
`mex`	Clears all MEX-functions from memory, except for locked functions or functions that are currently in use. `clear mex` also clears breakpoints and persistent variables.
`variables`	Clears all variables from the workspace.

If you name a variable `all`, `classes`, `functions`, `java`, `import`, or `variables`, calling `clear` followed by that name deletes the variable with that name. `clear` does not interpret the name as a keyword in this context. For example, if the workspace contains variables `a`, `all`, `b`, and `ball`, `clear all` removes the variable `all` only.

Tips

- The `clear` function can remove variables that you specify. To remove all but some specified variables, use `clearvars` instead.
- You can clear the handle of a figure or graphics object, but the object itself is not removed. Use `delete` to remove objects. Deleting an object does not delete the variable (if any) used for storing its handle.
- The `clear` function does not clear Simulink® models. Use `bdclose` instead.
- On UNIX systems, `clear` does not affect the amount of memory allocated to the MATLAB process.

Examples

Clear a Single Variable

Define two variables `a` and `b`, and then clear `a`.

```
a=1;
b=2;
clear a
whos
```

```
Name      Size            Bytes  Class     Attributes

b         1x1                 8  double
```

Only variable `b` remains in the workspace.

Clear Specific Variables

Using regular expressions, clear those variables with names that begin with `Mon`, `Tue`, or `Wed`.

```
clear -regexp ^Mon ^Tue ^Wed;
```

Clear All Variables and Functions

```
clear all
```

Clear All Compiled Scripts, Functions, and MEX-functions

```
clear functions
```

If a function is locked, it will not be cleared from memory.

See Also

`clc` | `clearvars` | `close` | `delete` | `import` | `inmem` | `load` | `mlock` | `persistent` | `whos` | `workspace`

Concepts

- "Base and Function Workspaces"
- "Strategies for Efficient Use of Memory"
- "Modifying and Reloading Classes"
- "The Java Class Path"
- "Regular Expressions"

Purpose

Clear variables from memory

Alternatives

As an alternative to the `clearvars` function, in the Workspace browser, select variables to clear and then press **Delete**.

Syntax

```
clearvars v1 v2 ...
clearvars -global
clearvars -global v1 v2 ...
clearvars -regexp p1 p2 ...
clearvars -except v1 v2 ...
clearvars -except -regexp p1 p2 ...
clearvars v1 v2 ... -except -regexp p1 p2 ...
clearvars -regexp p1 p2 ... -except v1 v2 ...
```

Description

`clearvars v1 v2 ...` clears variables `v1`, `v2`, and so on from the currently active workspace. Each input must be an unquoted string specifying the variable to be cleared. This string may include the wildcard character (`*`) to clear all variables that match a pattern. For example, `clearvars X*` clears all the variables in the current workspace that start with the letter `X`.

If any of the variables `v1`, `v2`, and so on, are global, `clearvars` removes these variables from the current workspace only, leaving them accessible to any functions that declare them as global.

`clearvars -global` removes all global variables, including those made global within functions.

`clearvars -global v1 v2 ...` completely removes the specified global variables.

The `-global` flag may be used with any of the following syntaxes. When used in this way, it must immediately follow the function name.

`clearvars -regexp p1 p2 ...` clears all variables that match regular expression patterns `p1`, `p2`, and so on.

`clearvars -except v1 v2 ...` clears all variables except for those specified following the `-except` flag. Use the wildcard character '*'

in a variable name to exclude variables that match a pattern from being cleared. `clearvars -except X*` clears all the variables in the current workspace, except for those that start with `X`, for instance. Use `clearvars -except` to keep the variables you want and remove all others.

`clearvars -except -regexp p1 p2 ...` clears all variables except those that regular expression patterns `p1`, `p2`. If used in this way, the `-regexp` flag must immediately follow the `-except` flag.

`clearvars v1 v2 ...  -except -regexp p1 p2 ...` can be used to specify variables to clear that do not match specified regular expression patterns.

`clearvars -regexp p1 p2 ...  -except v1 v2 ...` clears variables that match `p1`, `p2`, ..., except for variables `v1`, `v2`, ...

Examples

Clear variables starting with `a`, except for the variable `ab`:

```
clearvars a* -except ab
```

Clear all global variables except those starting with `x`:

```
clearvars -global -except x*
```

Clear variables that start with `b` and are followed by 3 digits, for the variable `b106`:

```
clearvars -regexp ^b\d{3}$ -except b106
```

Clear variables that start with `a`, except those ending with `a`:

```
clearvars a* -except -regexp a$
```

See Also

`clear` | `exist` | `global` | `persistent` | `save` | `who` | `whos`

How To

- “What Is the MATLAB Workspace?”

Purpose

Remove serial port object from MATLAB workspace

Syntax

`clear obj`

Description

`clear obj` removes `obj` from the MATLAB workspace, where `obj` is a serial port object or an array of serial port objects.

Tips

If `obj` is connected to the device and it is cleared from the workspace, then `obj` remains connected to the device. You can restore `obj` to the workspace with the `instrfind` function. A serial port object connected to the device has a `Status` property value of `open`.

To disconnect `obj` from the device, use the `fclose` function. To remove `obj` from memory, use the `delete` function. You should remove invalid serial port objects from the workspace with `clear`.

Examples

This example creates the serial port object `s` on a Windows platform, copies `s` to a new variable `scopy`, and clears `s` from the MATLAB workspace. `s` is then restored to the workspace with `instrfind` and is shown to be identical to `scopy`.

```
s = serial('COM1');
scopy = s;
clear s
s = instrfind;
isequal(scopy,s)
ans =
     1
```

See Also

`delete` | `fclose` | `instrfind` | `isvalid` | `Status`

clf

Purpose

Clear current figure window

Syntax

```
clf
clf('reset')
clf(fig)
clf(fig,'reset')
figure_handle = clf(...)
```

Description

`clf` deletes from the current figure all graphics objects whose handles are not hidden (i.e., their `HandleVisibility` property is set to `on`).

`clf('reset')` deletes from the current figure all graphics objects regardless of the setting of their `HandleVisibility` property and resets all figure properties except `Position`, `Units`, `PaperPosition`, and `PaperUnits` to their default values.

`clf(fig)` or `clf(fig,'reset')` clears the single figure with handle `fig`.

`figure_handle = clf(...)` returns the handle of the figure. This is useful when the figure `IntegerHandle` property is `off` because the noninteger handle becomes invalid when the reset option is used (i.e., `IntegerHandle` is reset to `on`, which is the default).

Tips

The `clf` command behaves the same way when issued on the command line as it does in callback routines — it does not recognize the `HandleVisibility` setting of `callback`. This means that when issued from within a callback routine, `clf` deletes only those objects whose `HandleVisibility` property is set to `on`.

Alternatives

Use **Clear Figure** from the figure window's **Edit** menu to clear the contents of a figure. You can also create a *desktop shortcut* to clear the current figure with one mouse click. See "Create Shortcuts to Rerun Commands".

See Also

`cla` | `clc` | `hold` | `reset`

Purpose

Copy and paste strings to and from system clipboard

Syntax

```
clipboard('copy', data)
str = clipboard('paste')
data = clipboard('pastespecial')
```

Description

`clipboard('copy', data)` sets the clipboard contents to `data`. If `data` is not a character array, the clipboard uses `mat2str` to convert `data` to a string.

`str = clipboard('paste')` returns the current contents of the clipboard as a string or, if MATLAB cannot convert the current clipboard contents to a string, as an empty string (`' '`).

`data = clipboard('pastespecial')` returns the current contents of the clipboard as an array by using `uiimport`.

Definitions

The `clipboard` function requires Oracle® Java software.

See Also

`load` | `mat2str` | `uiimport`

clock

Purpose

Current date and time as date vector

Syntax

```
c = clock
```

Description

`c = clock` returns a six-element date vector containing the current date and time in decimal form:

```
[year month day hour minute seconds]
```

Examples

Round clock Output to Integer Display

Use `clock` to return the current date and time.

```
format shortg
c = clock

c =
        2012           3          27          13          58
```

The sixth element of the date vector output (seconds) is accurate to several digits beyond the decimal point.

Use the `fix` function to round to integer display format.

```
fix(c)

ans =
        2012           3          27          13          58           9
```

Tips

- To time the duration of an event, use the `tic` and `toc` functions instead of `clock` and `etime`. The `clock` function is based on the system time, which can be adjusted periodically by the operating system, and thus might not be reliable in time comparison operations.

See Also

`date` | `now` | `tic` | `toc` | `cputime` | `etime` | `fix`

Purpose

Remove specified figure

Syntax

```
close
close(h)
close name
close all
close all hidden
close all force
status = close(...)
```

Description

`close` deletes the current figure or the specified figure(s). It optionally returns the status of the close operation.

`close` deletes the current figure (equivalent to `close(gcf)`).

`close(h)` deletes the figure identified by `h`. If `h` is a vector or matrix, `close` deletes all figures identified by `h`.

`close name` deletes the figure with the specified name.

`close all` deletes all figures whose handles are not hidden.

`close all hidden` deletes all figures including those with hidden handles.

`close all force` deletes all figures, including GUIs for which `CloseRequestFcn` has been altered to not close the window.

`status = close(...)` returns `1` if the specified windows have been deleted and `0` otherwise.

Algorithms

The `close` function works by evaluating the specified figure's `CloseRequestFcn` property with the statement

```
eval(get(h,'CloseRequestFcn'))
```

The default `CloseRequestFcn`, `closereq`, deletes the current figure using `delete(get(0,'CurrentFigure'))`. If you specify multiple figure handles, `close` executes each figure's `CloseRequestFcn` in turn. If an error that terminates the execution of a `CloseRequestFcn` occurs, the

figure is not deleted. Note that using your computer's window manager (i.e., the **Close** menu item) also calls the figure's `CloseRequestFcn`.

If a figure's handle is hidden (i.e., the figure's `HandleVisibility` property is set to `callback` or `off` and the root `ShowHiddenHandles` property is set to `on`), you must specify the `hidden` option when trying to access a figure using the `all` option.

To delete all figures unconditionally, use the statements

```
set(0,'ShowHiddenHandles','on')
delete(get(0,'Children'))
```

The figure `CloseRequestFcn` allows you to either delay or abort the closing of a figure once the `close` function has been issued. For example, you can display a dialog box to see if the user really wants to delete the figure or save and clean up before closing.

When coding a `CloseRequestFcn` callback, make sure that it does not call `close`, because this sets up a recursion that results in a MATLAB warning. Instead, the callback should destroy the figure with `delete`. The `delete` function does not execute the figure's `CloseRequestFcn`; it simply deletes the specified figure.

See Also

`delete` | `figure` | `gcf` | Figure: `HandleVisibility` | Root: `ShowHiddenHandles`

Purpose

Close Tiff object

Syntax

`tiffobj.close()`

Description

`tiffobj.close()` closes a Tiff object.

Examples

Open a Tiff object and then close it.

```
% Replace 'myfile.tif' with a TIFF file on your MATLAB path.
t = Tiff('myfile.tif', 'w');
%
% Close Tiff object.
t.close();
```

References

This method corresponds to the `TIFFClose` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTiff - TIFF Library and Utilities`.

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

close (avifile)

Purpose

Close Audio/Video Interleaved (AVI) file

Syntax

`aviobj = close(aviobj)`

Description

`aviobj = close(aviobj)` finishes writing and closes the AVI file associated with `aviobj`, which is an AVI file object created using the `avifile` function.

To close all open AVI files, use the `clear mex` command.

See Also

`avifile` | `addframe (avifile)` | `movie2avi`

Purpose

Close connection to FTP server

Syntax

```
close(ftpobj)
```

Description

`close(ftpobj)` closes the connection to the FTP server.

Tips

- If you do not run `close` at the end of your session, the connection either times out automatically or terminates when you exit MATLAB.
- After calling `close`, calling any other FTP method on the same object automatically reopens the connection.
- `close` does not return any output to indicate success or failure.

Input Arguments

ftpobj

FTP object created by `ftp`.

Examples

Connect to the MathWorks FTP server, and then disconnect:

```
mw=ftp('ftp.mathworks.com');
close(mw)
```

See Also

`ftp`

VideoWriter.close

Purpose

Close file after writing video data

Syntax

```
close(writerObj)
```

Description

`close(writerObj)` closes the file associated with object `writerObj`. The object remains in the workspace. If you call the `open` method after closing, `open` discards any existing contents of the file.

Input Arguments

writerObj

`VideoWriter` object created by the `VideoWriter` function.

Examples

AVI File from Animation

Write a sequence of frames to a compressed AVI file, `peaks.avi`.

Prepare the new file.

```
writerObj = VideoWriter('peaks.avi');
open(writerObj);
```

Generate initial data and set axes and figure properties.

```
Z = peaks; surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
set(gcf,'Renderer','zbuffer');
```

Setting the `Renderer` property to `zbuffer` or `Painters` works around limitations of `getframe` with the OpenGL® renderer on some Windows systems.

Create a set of frames and write each frame to the file.

```
for k = 1:20
   surf(sin(2*pi*k/20)*Z,Z)
   frame = getframe;
   writeVideo(writerObj,frame);
```

```
end

close(writerObj);
```

See Also

writeVideo | VideoWriter | open

closereq

Purpose

Default figure close request function

Syntax

```
closereq
```

Description

`closereq` deletes the current figure.

See Also

| CloseRequestFcn

Purpose

Name of source control system

Syntax

```
cmopts
```

Description

`cmopts` returns the name of your version control system.

Output Arguments

Value Returned by cmopts	Description	Platform Supported On
`clearcase`	ClearCase® software from IBM® Rational®	UNIX platforms
`customverctrl`	Custom interface created using `customverctrl` function	UNIX platforms
`cvs`	Concurrent Version System (CVS)	UNIX platforms
`none`	No source control system selected	
`pvcs`	PVCS® and ChangeMan® software	UNIX platforms
`rcs`	Revision Control System (RCS)	UNIX platforms
Any SCC-compliant source control system, for example, `Microsoft Visual SourceSafe`	Varies	Windows platforms

Alternatives

To view the currently selected source control system, click the **Preferences** button on the **Home** tab, and select **General > Source Control**.

See Also

`checkin` | `checkout` | `customverctrl` | `undocheckout` | `verctrl`

How To

- "Source Control Interface on Microsoft Windows"
- "Source Control Interface on UNIX Platforms"

Purpose

Rearrange colors in colormap

Syntax

```
[Y,newmap] = cmpermute(X,map)
[Y,newmap] = cmpermute(X,map,index)
```

Description

`[Y,newmap] = cmpermute(X,map)` randomly reorders the colors in `map` to produce a new colormap, `newmap`. The `cmpermute` function also modifies the values in `X` to maintain correspondence between the indices and the colormap, and returns the result in `Y`. The image `Y` and associated colormap, `newmap`, produce the same image as `X` and `map`.

`[Y,newmap] = cmpermute(X,map,index)` uses an ordering matrix (such as the second output of `sort`) to define the order of colors in the new colormap.

Class Support

The input image `X` can be of class `uint8` or `double`. `Y` is returned as an array of the same class as `X`.

Examples

Randomly reorder a colormap and compare the two images:

```
load trees
image(X)
[Y, newmap] = cmpermute(X,map);
figure
image(Y)
```

See Also

`randperm` | `sort`

cmunique

Purpose

Eliminate duplicate colors in colormap; convert grayscale or truecolor image to indexed image

Syntax

```
[Y,newmap] = cmunique(X,map)
[Y,newmap] = cmunique(RGB)
[Y,newmap] = cmunique(I)
```

Description

`[Y,newmap] = cmunique(X,map)` returns the indexed image `Y` and associated colormap, `newmap`, that produce the same image as `(X,map)` but with the smallest possible colormap. The `cmunique` function removes duplicate rows from the colormap and adjusts the indices in the image matrix accordingly.

`[Y,newmap] = cmunique(RGB)` converts the truecolor image `RGB` to the indexed image `Y` and its associated colormap, `newmap`. The return value `newmap` is the smallest possible colormap for the image, containing one entry for each unique color in `RGB`.

Note `newmap` might be very large, because the number of entries can be as many as the number of pixels in `RGB`.

`[Y,newmap] = cmunique(I)` converts the grayscale image `I` to an indexed image `Y` and its associated colormap, `newmap`. The return value, `newmap`, is the smallest possible colormap for the image, containing one entry for each unique intensity level in `I`.

Class Support

The input image can be of class `uint8`, `uint16`, or `double`. The class of the output image `Y` is `uint8` if the length of `newmap` is less than or equal to 256. If the length of `newmap` is greater than 256, `Y` is of class `double`.

Examples

1. Use the magic function to create a sample 4-by-4 image that uses every value in the range between 1 and 16.

   ```
   X = magic(4)
   ```

```
X =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

2 Concatenate two 8-entry grayscale colormaps created using the `gray` function. The resultant colormap, `map`, has 16 entries. Entries 9 through 16 are duplicates of entries 1 through 8.

```
map = [gray(8); gray(8)]
size(map)

ans =

    16     3
```

3 Use `cmunique` to eliminate duplicate entries in the colormap.

```
[Y, newmap] = cmunique(X, map);
size(newmap)

ans =

     8     3
```

`cmunique` adjusts the values in the original image `X` to index the new colormap.

```
Y =

     7     1     2     4
     4     2     1     7
     0     6     5     3
     3     5     6     0
```

4 View both images to verify that their appearance is the same.

```
figure, imshow(X, map, 'InitialMagnification', 'fit')
figure, imshow(Y, newmap, 'InitialMagnification', 'fit')
```

See Also

`rgb2ind`

Purpose

Column approximate minimum degree permutation

Syntax

```
p = colamd(S)
```

Description

`p = colamd(S)` returns the column approximate minimum degree permutation vector for the sparse matrix `S`. For a non-symmetric matrix `S`, `S(:,p)` tends to have sparser LU factors than `S`. The Cholesky factorization of `S(:,p)' * S(:,p)` also tends to be sparser than that of `S'*S`.

`knobs` is a two-element vector. If S is `m`-by-`n`, then rows with more than `(knobs(1))*n` entries are ignored. Columns with more than `(knobs(2))*m` entries are removed prior to ordering, and ordered last in the output permutation `p`. If the `knobs` parameter is not present, then `knobs(1)` = `knobs(2) = spparms('wh_frac')`.

`stats` is an optional vector that provides data about the ordering and the validity of the matrix `S`.

`stats(1)`	Number of dense or empty rows ignored by `colamd`
`stats(2)`	Number of dense or empty columns ignored by `colamd`
`stats(3)`	Number of garbage collections performed on the internal data structure used by `colamd` (roughly of size `2.2*nnz(S) + 4*m + 7*n` integers)
`stats(4)`	`0` if the matrix is valid, or `1` if invalid
`stats(5)`	Rightmost column index that is unsorted or contains duplicate entries, or `0` if no such column exists

`stats(6)`	Last seen duplicate or out-of-order row index in the column index given by `stats(5)`, or `0` if no such row index exists
`stats(7)`	Number of duplicate and out-of-order row indices

Although MATLAB built-in functions generate valid sparse matrices, a user may construct an invalid sparse matrix using the MATLAB C or Fortran APIs and pass it to `colamd`. For this reason, `colamd` verifies that `S` is valid:

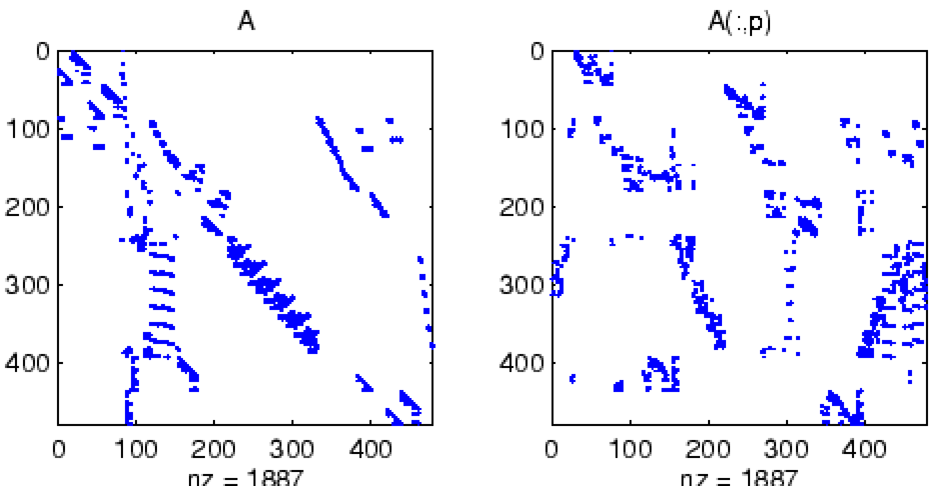
- If a row index appears two or more times in the same column, `colamd` ignores the duplicate entries, continues processing, and provides information about the duplicate entries in `stats(4:7)`.
- If row indices in a column are out of order, `colamd` sorts each column of its internal copy of the matrix `S` (but does not repair the input matrix `S`), continues processing, and provides information about the out-of-order entries in `stats(4:7)`.
- If `S` is invalid in any other way, `colamd` cannot continue. It prints an error message, and returns no output arguments (`p` or `stats`) .

The ordering is followed by a column elimination tree post-ordering.

Examples

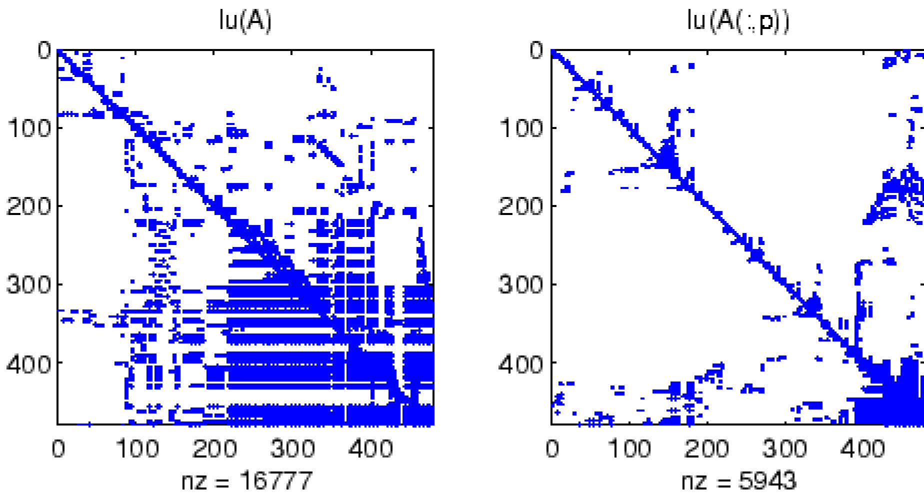
The Harwell-Boeing collection of sparse matrices and the MATLAB demos directory include a test matrix `west0479`. It is a matrix of order 479 resulting from a model due to Westerberg of an eight-stage chemical distillation column. The spy plot shows evidence of the eight stages. The `colamd` ordering scrambles this structure.

```
load west0479
A = west0479;
p = colamd(A);
subplot(1,2,1), spy(A,4), title('A')
subplot(1,2,2), spy(A(:,p),4), title('A(:,p)')
```



Comparing the spy plot of the LU factorization of the original matrix with that of the reordered matrix shows that minimum degree reduces the time and storage requirements by better than a factor of 2.8. The nonzero counts are 16777 and 5904, respectively.

```
spy(lu(A),4)
spy(lu(A(:,p)),4)
```



References

[1] The authors of the code for “colamd” are Stefan I. Larimore and Timothy A. Davis (davis@cise.ufl.edu), University of Florida. The algorithm was developed in collaboration with John Gilbert, Xerox PARC, and Esmond Ng, Oak Ridge National Laboratory. Sparse Matrix Algorithms Research at the University of Florida: `http://www.cise.ufl.edu/research/sparse/`

See Also

`colperm` | `spparms` | `symamd` | `symrcm`

Purpose

Colorbar showing color scale

Syntax

```
colorbar
colorbar('off')
colorbar('hide')
colorbar('delete')
colorbar(...,'peer',axes_handle)
colorbar(...,'location')
colorbar(...,'PropertyName',propertyvalue)
cbar_axes = colorbar(...)
colorbar('option')
colorbar(cbar_handle,'option')
```

Description

The `colorbar` function displays the current colormap in the current figure and resizes the current axes to accommodate the colorbar.

`colorbar` adds a new vertical colorbar on the right side of the current axes. If a colorbar exists in that location, `colorbar` replaces it with a new one. If a colorbar exists at a nondefault location, it is retained along with the new colorbar.

`colorbar('off')`, `colorbar('hide')`, and `colorbar('delete')` delete all colorbars associated with the current axes.

`colorbar(...,'peer',axes_handle)` creates a colorbar associated with the axes `axes_handle` instead of the current axes.

`colorbar(...,'location')` adds a colorbar in the specified orientation with respect to the axes. If a colorbar exists at the location specified, it is replaced. Any colorbars not occupying the specified location are retained. Possible values for *location* are

North	Inside plot box near top
South	Inside bottom
East	Inside right
West	Inside left

`NorthOutside`	Outside plot box near top
`SouthOutside`	Outside bottom
`EastOutside`	Outside right
`WestOutside`	Outside left

Using one of the `...Outside` values for *`location`* ensures that the colorbar does not overlap the plot, whereas overlaps can occur when you specify any of the other four values.

`colorbar(...,'`*`PropertyName`*`',propertyvalue)` specifies property names and values for the axes object used to create the colorbar. See Axes Properties for a description of the properties you can set. The *`location`* property applies only to colorbars and legends, not to axes.

`cbar_axes = colorbar(...)` returns a handle to a new colorbar object, which is a child of the current figure. If a colorbar exists, a new one is still created.

`colorbar('`*`option`*`')` specifies options for colorbar visibility. '*`option`*' is one of `'off'`, `` `hide' `` or `` `delete' ``. All of the options delete the colorbar in the current axes.

`colorbar(cbar_handle,'`*`option`*`')` specifies options for colorbar visibility. `'`*`option`*`'` is one of `'off'`, `` `hide' `` or `` `delete' ``. All of the options delete the colorbar specified by the `cbar_handle`.

You can use `colorbar` with 2-D and 3-D plots.

Tips

To obtain the handle to an existing colorbar, use the command

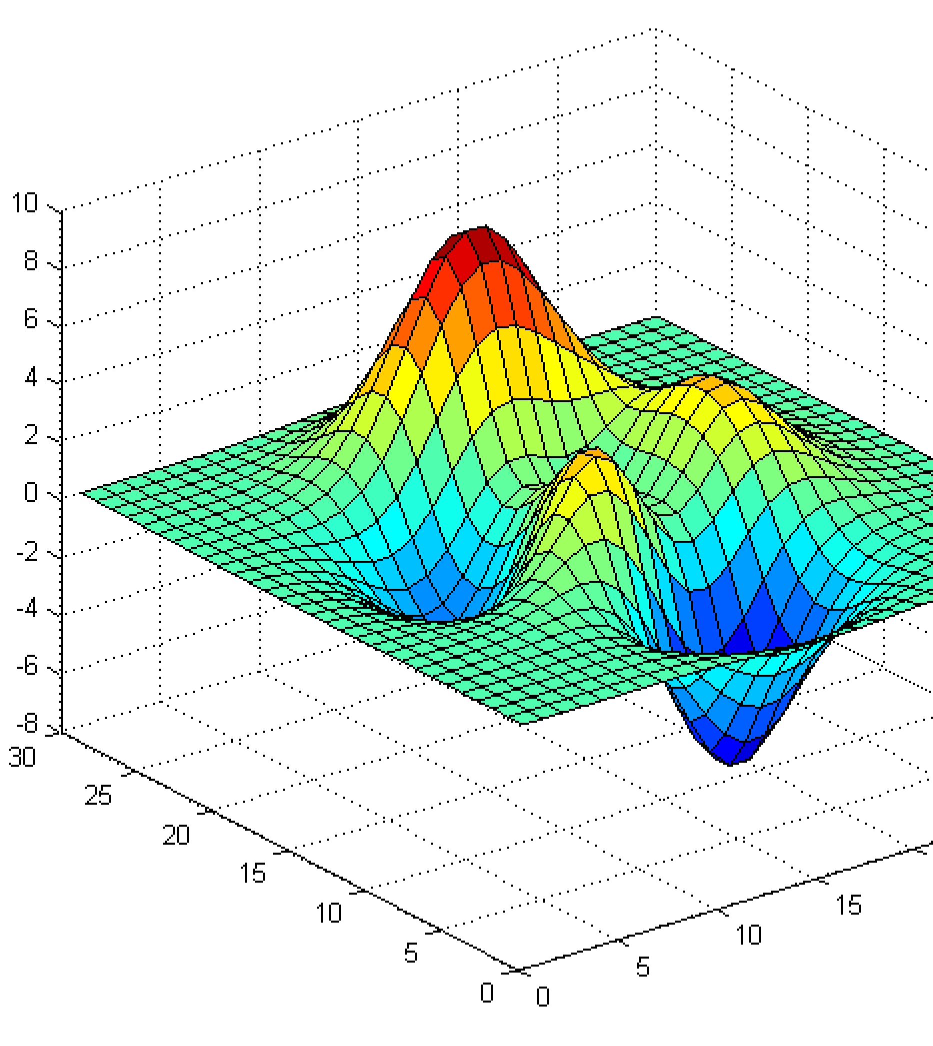
```
cbar_handle = findobj(figure_handle,'tag','Colorbar')
```

where `figure_handle` is the handle of the figure containing the colorbar you want to modify. If the figure contains more than one colorbar, `cbar_handle` is returned as a vector, and you must choose which of the handles to specify to `colorbar`.

Examples

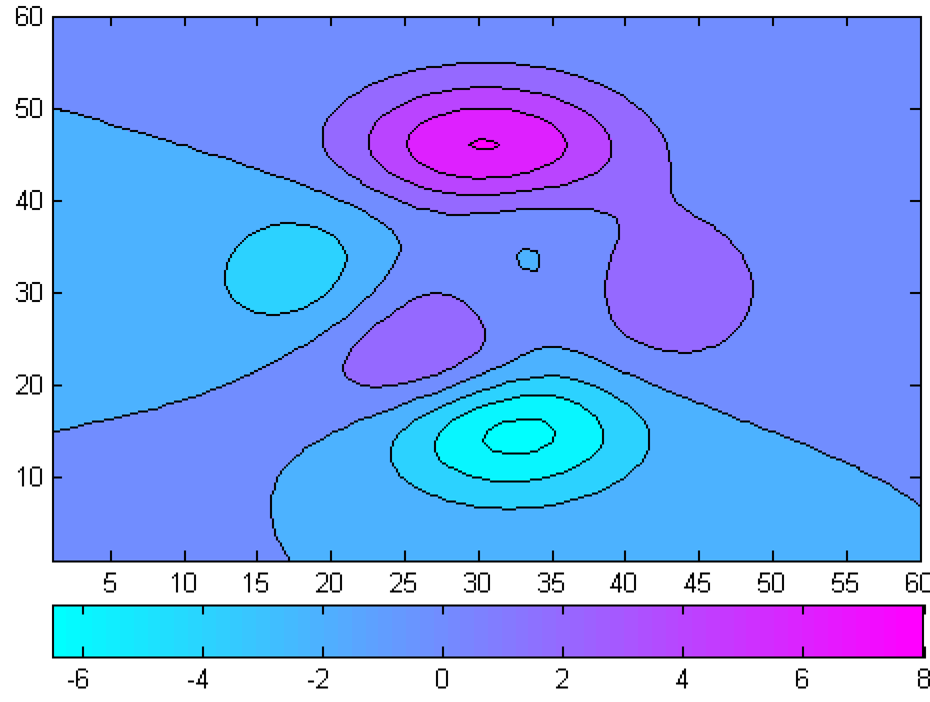
Display a colorbar beside the axes and use descriptive text strings as *y*-tick labels. Labels repeat cyclically when the number of *y*-ticks is greater than the number of labels, and not all labels will appear if there are fewer *y*-ticks than labels you have specified. Also note that when colorbars are horizontal, their ticks and labels are governed by the `XTick` property rather than the `YTick` property. For more information, see "Labeling Colorbar Ticks".

```
surf(peaks(30))
colorbar('YTickLabel',...
    {'Freezing','Cold','Cool','Neutral',...
     'Warm','Hot','Burning','Nuclear'})
```


10
8
6
4
2
0
-2
-4
-6
-8
30
25
20
15
10
5
0
0
5
10
15

Display a horizontal colorbar beneath the axes of a filled contour plot:

```
contourf(peaks(60))
colormap cool
colorbar('location','southoutside')
```



Alternatives

Add a colorbar to a plot with the colorbar tool on the figure toolbar, or use **Insert > Colorbar** from the figure menu. Use the Property

Editor to modify the position, font and other properties of a legend. .
For details, see “Working in Plot Edit Mode”.

See Also

`colormap`

Purpose

Set default property values to display different color schemes

Syntax

```
colordef white
colordef black
colordef none
colordef(fig,color_option)
h = colordef('new',color_option)
```

Description

`colordef` enables you to select either a white or black background for graphics display. It sets axis lines and labels so that they contrast with the background color.

`colordef white` sets the axis background color to white, the axis lines and labels to black, and the figure background color to light gray.

`colordef black` sets the axis background color to black, the axis lines and labels to white, and the figure background color to dark gray.

`colordef none` sets the figure coloring to that used by MATLAB Version 4. The most noticeable difference is that the axis background is set to `'none'`, making the axis background and figure background colors the same. The figure background color is set to black.

`colordef(fig,`*`color_option`*`)` sets the color scheme of the figure identified by the handle `fig` to one of the color options `'white'`, `'black'`, or `'none'`. When you use this syntax to apply `colordef` to an existing figure, the figure must have no graphic content. If it does, you should first clear it (via `clf`) before using this form of the command.

`h = colordef('new',`*`color_option`*`)` returns the handle to a new figure created with the specified color options (i.e., `'white'`, `'black'`, or `'none'`). This form of the command is useful for creating GUIs when you may want to control the default environment. The figure is created with `'visible','off'` to prevent flashing.

Tips

`colordef` affects only subsequently drawn figures, not those currently on the display. This is because `colordef` works by setting default property values (on the root or figure level). You can list the currently set default values on the root level with the statement

```
get(0,'defaults')
```

You can remove all default values using the `reset` command:

```
reset(0)
```

See the `get` and `reset` references pages for more information.

See Also

`whitebg` | `clf`

Purpose

Set and get current colormap

Syntax

```
colormap(map)
colormap('default')
cmap = colormap
colormap(ax,...)
```

Description

A colormap is an m-by-3 matrix of real numbers between 0.0 and 1.0. Each row is an RGB vector that defines one color. The kth row of the colormap defines the kth color, where `map(k,:) = [r(k) g(k) b(k)])` specifies the intensity of red, green, and blue.

`colormap(map)` sets the colormap to the matrix `map`. If any values in `map` are outside the interval [0 1], you receive the error `Colormap must have values in [0,1]`.

`colormap('default')` sets the current colormap to the default colormap.

`cmap = colormap` retrieves the current colormap. The values returned are in the interval [0 1].

`colormap(ax,...)` uses the figure corresponding to axes `ax` instead of the current figure.

Specifying Colormaps

Files in the `color` folder generate a number of colormaps. Each file accepts the colormap size as an argument. For example,

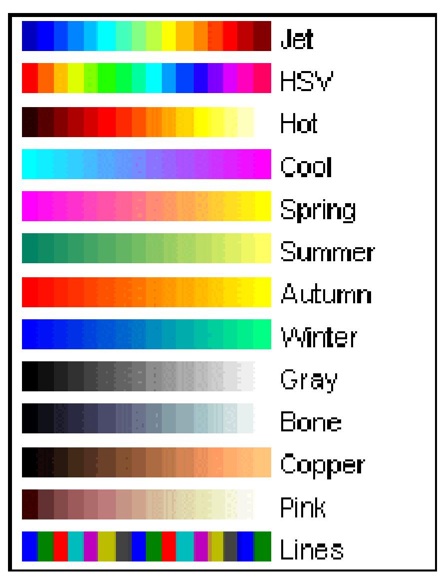
```
colormap(hsv(128))
```

creates an `hsv` colormap with 128 colors. If you do not specify a size, a colormap the same size as the current colormap is created.

Supported Colormaps

The built-in MATLAB colormaps are illustrated and described below. In addition to specifying built-in colormaps programmatically, you can use the **Colormap** menu in the **Figure Properties** pane of the Plot Tools GUI to select one interactively.

The named built-in colormaps are the following:



- `autumn` varies smoothly from red, through orange, to yellow.
- `bone` is a grayscale colormap with a higher value for the blue component. This colormap is useful for adding an "electronic" look to grayscale images.
- `colorcube` contains as many regularly spaced colors in RGB color space as possible, while attempting to provide more steps of gray, pure red, pure green, and pure blue.
- `cool` consists of colors that are shades of cyan and magenta. It varies smoothly from cyan to magenta.
- `copper` varies smoothly from black to bright copper.
- `flag` consists of the colors red, white, blue, and black. This colormap completely changes color with each index increment.
- `gray` returns a linear grayscale colormap.

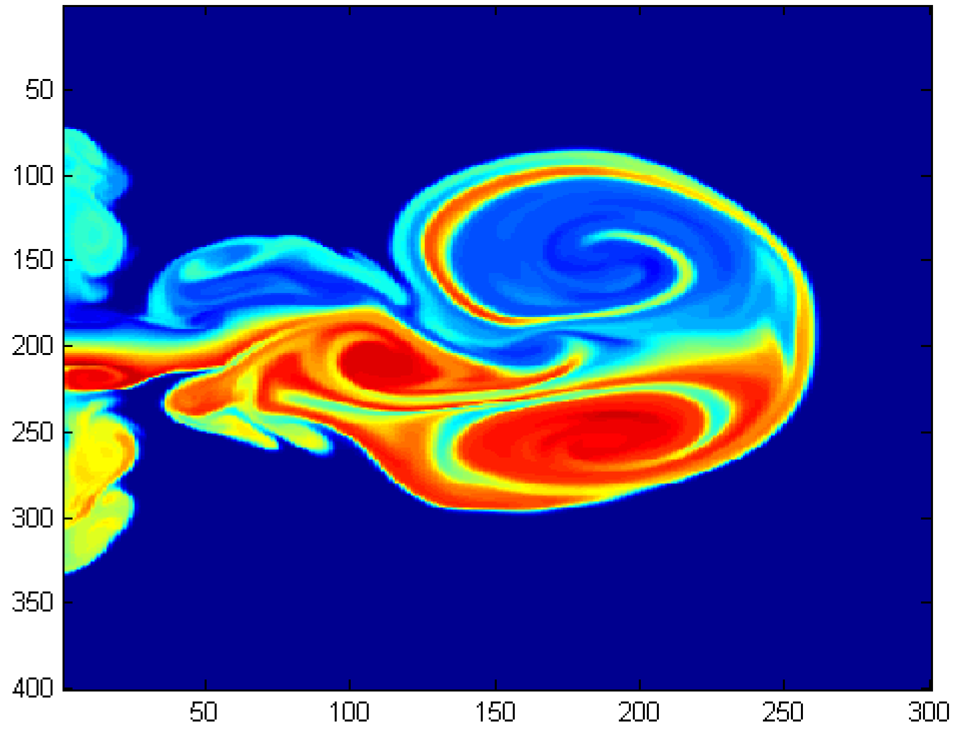
- `hot` varies smoothly from black through shades of red, orange, and yellow, to white.
- `hsv` varies the hue component of the hue-saturation-value color model. The colors begin with red, pass through yellow, green, cyan, blue, magenta, and return to red. The colormap is particularly appropriate for displaying periodic functions. `hsv(m)` is the same as `hsv2rgb([h ones(m,2)])` where `h` is the linear ramp, `h = (0:m 1)'/m`.
- `jet` ranges from blue to red, and passes through the colors cyan, yellow, and orange. It is a variation of the `hsv` colormap. The `jet` colormap is associated with an astrophysical fluid jet simulation from the National Center for Supercomputer Applications. See "Examples" on page 1-977.
- `lines` produces a colormap of colors specified by the axes `ColorOrder` property and a shade of gray.
- `pink` contains pastel shades of pink. The pink colormap provides sepia tone colorization of grayscale photographs.
- `prism` repeats the six colors red, orange, yellow, green, blue, and violet.
- `spring` consists of colors that are shades of magenta and yellow.
- `summer` consists of colors that are shades of green and yellow.
- `white` is an all white monochrome colormap.
- `winter` consists of colors that are shades of blue and green.

Examples

The `rgbplot` function plots colormap values. Try `rgbplot(hsv)`, `rgbplot(gray)`, and `rgbplot(hot)`.

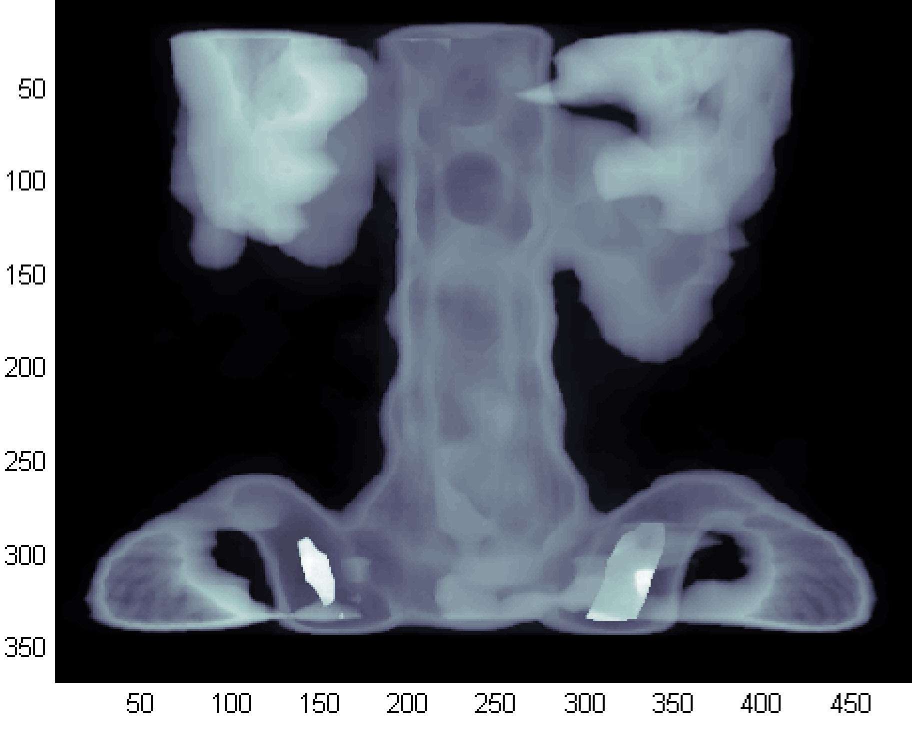
The following commands display the `flujet` data using the `jet` colormap:

```
load flujet
image(X)
colormap(jet)
```



The `demos` folder contains a CAT scan image of a human spine. Use the `image` function to view the data:

```
load spine
image(X)
colormap bone
```



Algorithms

Each figure has its own `colormap` property. `colormap` is a function that sets and gets this property.

Alternatives

Select a built-in colormap with the Property Editor. To modify the current colormap, use the Colormap Editor, accessible from **Edit > Colormap** on the figure menu.

See Also

`brighten` | `caxis` | `colorbar` | `colormapeditor` | `contrast` | `hsv2rgb` | `pcolor` | `rgbplot` | `rgb2hsv` | `Colormap`

How To

- “Coloring Mesh and Surface Plots”

Purpose

Open colormap editor

Syntax

```
colormapeditor
```

Description

`colormapeditor` displays the current figure's colormap as a strip of rectangular cells in the colormap editor. Node pointers are colored cells below the colormap strip that indicate points in the colormap where the rate of the variation of R, G, and B values changes. You can also work in the HSV colorspace by setting the **Interpolating Colorspace** selector to HSV.

You can also start the colormap editor by selecting **Colormap** from the **Edit** menu.

Node Pointer Operations

You can select and move node pointers to change a range of colors in the colormap. The color of a node pointer remains constant as you move it, but the colormap changes by linearly interpolating the RGB values between nodes.

Change the color at a node by double-clicking the node pointer. A color picker box appears, from which you can select a new color. After you select a new color at a node, the colors between nodes are reinterpolated.

You can select a different color map using the **Standard Colormaps** submenu of the GUI **Tools** menu. The Plotting Tools Property Editor has a dropdown menu that also lets you select from standard colormaps, but does not help you to modify a colormap.

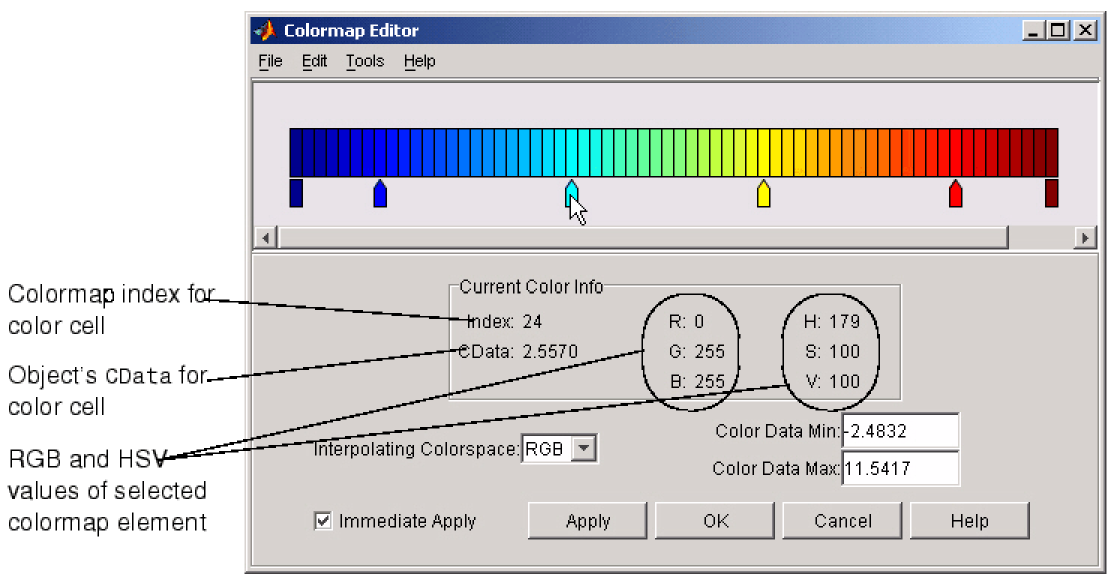
Operation	How to Perform
Select a built-in colormap	**Tools > Standard Colormaps**
Add a node	Click below the corresponding cell in the colormap strip.
Select a node	Left-click the node.

Operation	How to Perform
Select multiple nodes	Adjacent: left-click first node, **Shift+click** the last node. Nonadjacent: left-click first node, **Ctrl+click** subsequent nodes.
Move a node	Select and drag with the mouse or select and use the left and right arrow keys.
Move multiple nodes	Select multiple nodes and use the left and right arrow keys to move nodes as a group. Movement stops when one of the selected nodes hits an unselected node or an end node.
Delete a node	Select the node and then press the **Delete** key, or select **Delete** from the **Edit** menu, or type **Ctrl+x**.
Delete multiple nodes	Select the nodes and then press the **Delete** key, or select **Delete** from the **Edit** menu, or type **Ctrl+x**.
Display color picker for a node	Double-click the node pointer.

Current Color Info

When you put the mouse over a color cell or node pointer, the colormap editor displays the following information about that colormap element:

- The element's index in the colormap
- The value from the graphics object color data that is mapped to the node's color (i.e., data from the `CData` property of any image, patch, or surface objects in the figure)
- The color's RGB and HSV color value



Interpolating Colorspace

The colorspace determines what values are used to calculate the colors of cells between nodes. For example, in the RGB colorspace, internode colors are calculated by linearly interpolating the red, green, and blue intensity values from one node to the next. Switching to the HSV colorspace causes the colormap editor to recalculate the colors between nodes using the hue, saturation, and value components of the color definition.

Note that when you switch from one colorspace to another, the color editor preserves the number, color, and location of the node pointers, which can cause the colormap to change.

Interpolating in HSV. Since hue is conceptually mapped about a color circle, the interpolation between hue values can be ambiguous. To minimize this ambiguity, the interpolation uses the shortest distance around the circle. For example, interpolating between two nodes, one with hue of 2 (slightly orange red) and another with a hue of 356 (slightly magenta red), does not result in hues 3,4,5...353,354,355 (orange/red-yellow-green-cyan-blue-magenta/red). Taking the shortest distance around the circle gives 357,358,1,2 (orange/red-red-magenta/red).

Color Data Min and Max

The **Color Data Min** and **Color Data Max** text fields enable you to specify values for the axes `CLim` property. These values change the mapping of object color data (the `CData` property of images, patches, and surfaces) to the colormap. See "Axes Color Limits — the CLim Property" for discussion and examples of how to use this property.

Examples

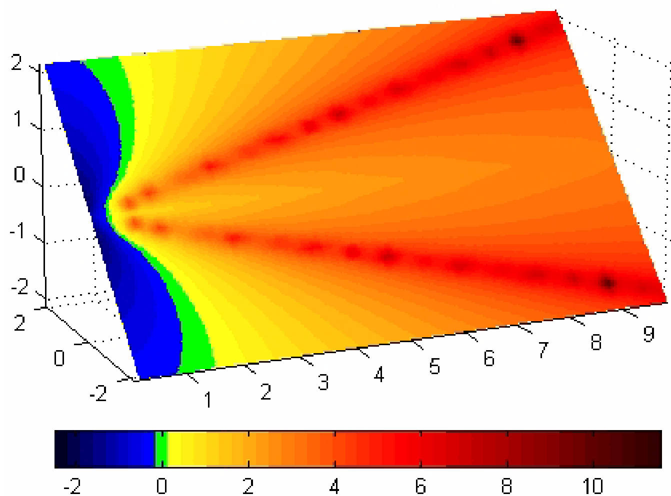
This example modifies a default MATLAB colormap so that ranges of data values are displayed in specific ranges of color. The graph is a slice plane illustrating a cross section of fluid flow through a jet nozzle. See the `slice` reference page for more information on this type of graph.

Example Objectives

The objectives are as follows:

- Regions of flow from left to right (positive data) are mapped to colors from yellow through orange to dark red. Yellow is slowest and dark red is the fastest moving fluid.
- Regions that have a speed close to zero are colored green.
- Regions where the fluid is actually moving right to left (negative data) are shades of blue (darker blue is faster).

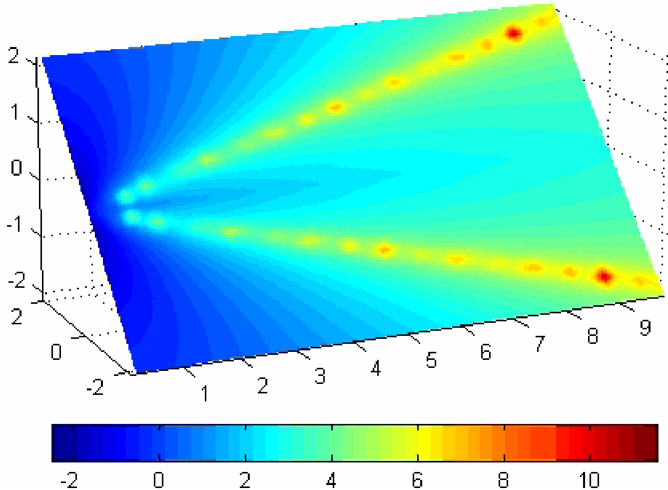
The following picture shows the desired coloring of the slice plane. The colorbar shows the data to color mapping.



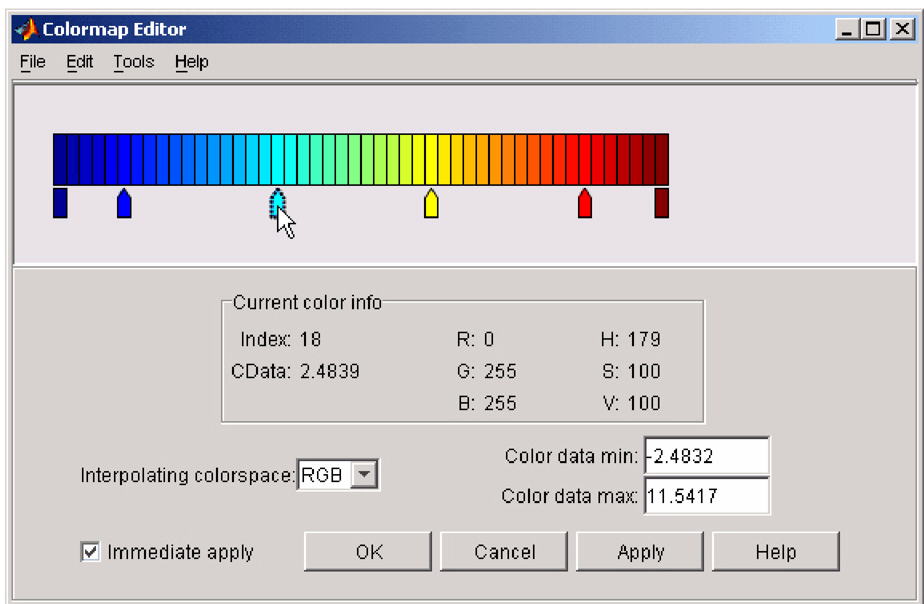
Running the Example

Note If you are viewing this documentation in the MATLAB help browser, you can display the graph used in this example by running this file from the MATLAB editor (select **Run** from the **Debug** menu).

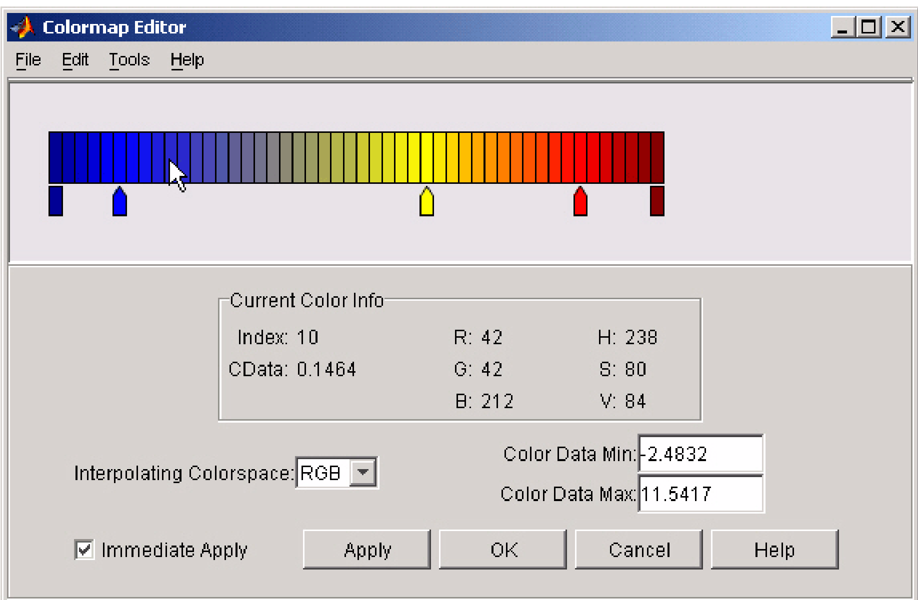
Initially, the default colormap (`jet`) colored the slice plane, as illustrated in the following picture. Note that this example uses a colormap that is 48 elements to display wider bands of color (the default is 64 elements).



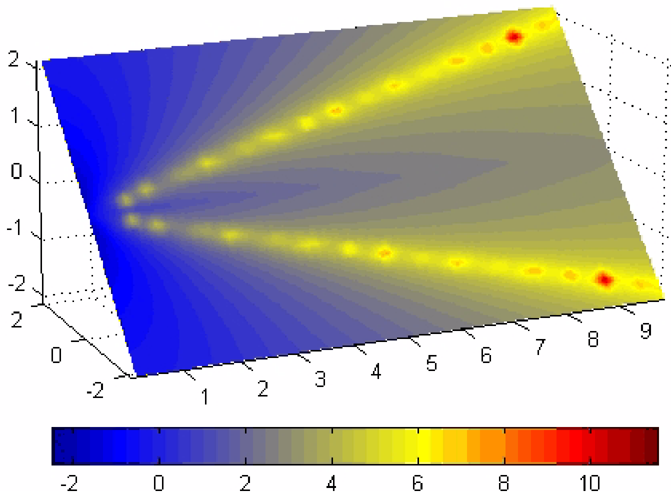
1 Start the colormap editor using the `colormapeditor` command. The color map editor displays the current figure' s colormap, as shown in the following picture.



2 Since we want the regions of left-to-right flow (positive speed) to range from yellow to dark red, we can delete the cyan node pointer. To do this, first select it by clicking with the left mouse button and press **Delete**. The colormap now looks like this.

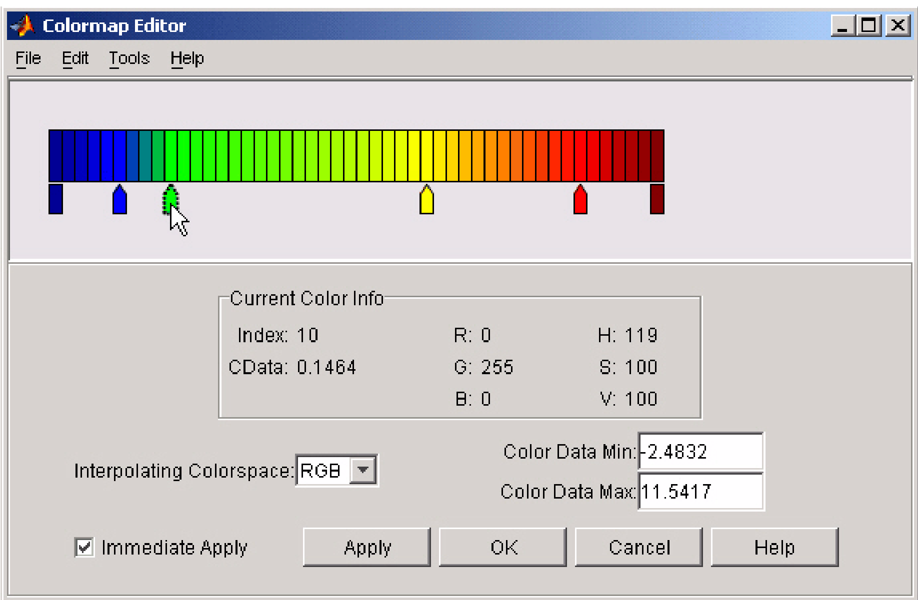


The **Immediate Apply** box is checked, so the graph displays the results of the changes made to the colormap.

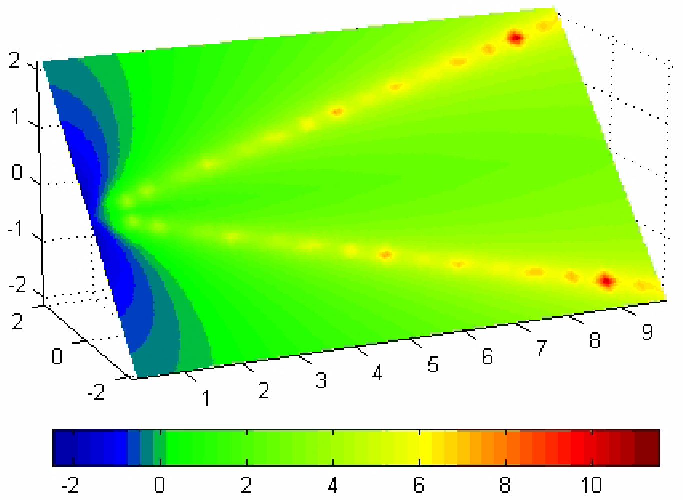


3 We want the fluid speed values around zero to stand out, so we need to find the color cell where the negative-to-positive transition occurs. Dragging the cursor over the color strip enables you to read the data values in the **Current Color Info** panel.

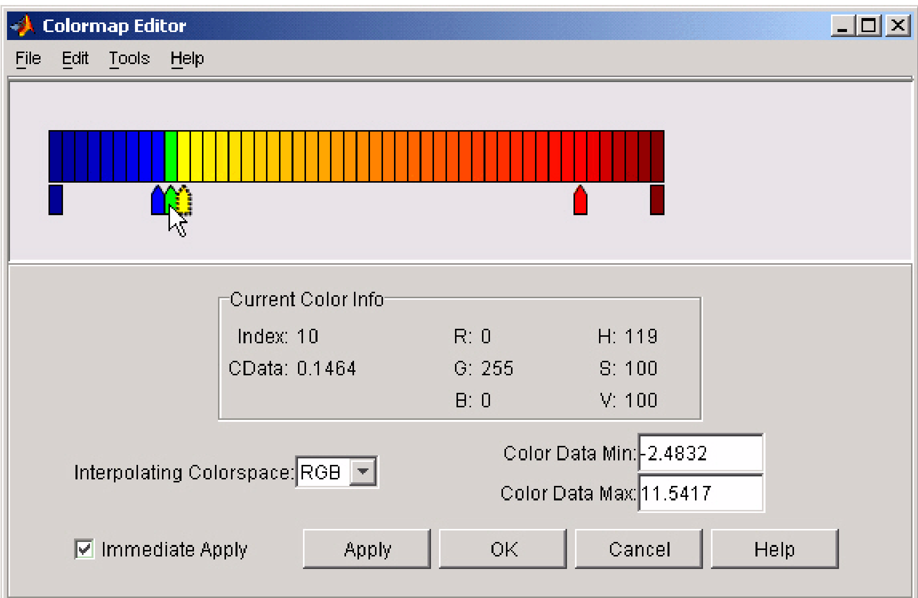
In this case, cell 10 is the first positive value, so we click below that cell and create a node pointer. Double-clicking the node pointer displays the color picker. Set the color of this node to green.



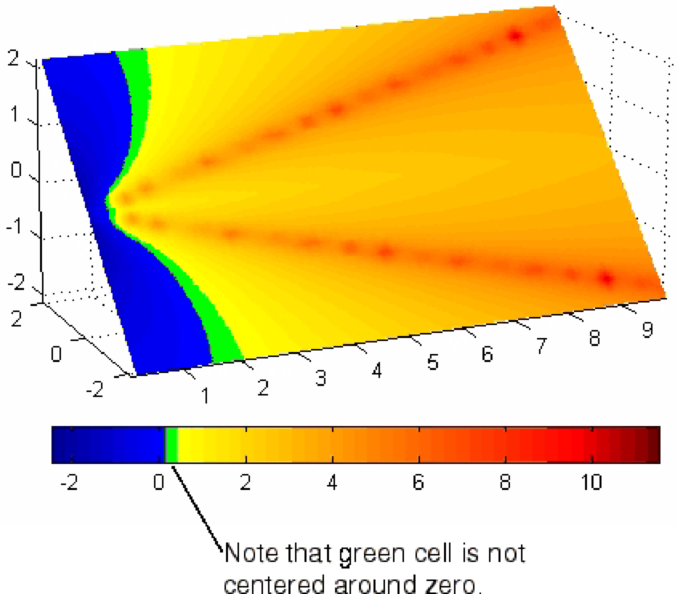
The graph continues to update to the modified colormap.



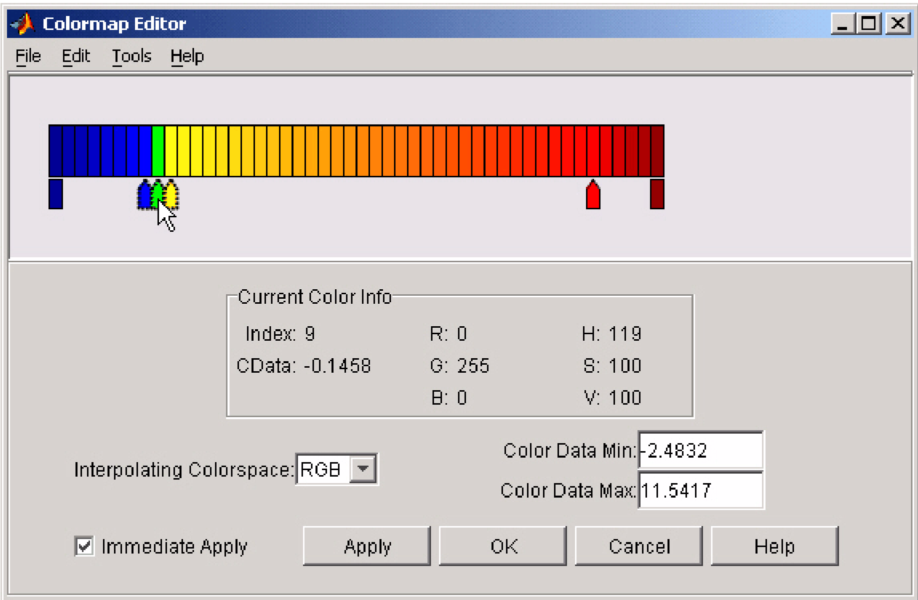
4 In the current state, the colormap colors are interpolated from the green node to the yellowish node about 20 cells away. We actually want only the single cell that is centered around zero to be colored green. To limit the color green to one cell, move the blue and yellow node pointers next to the green pointer.



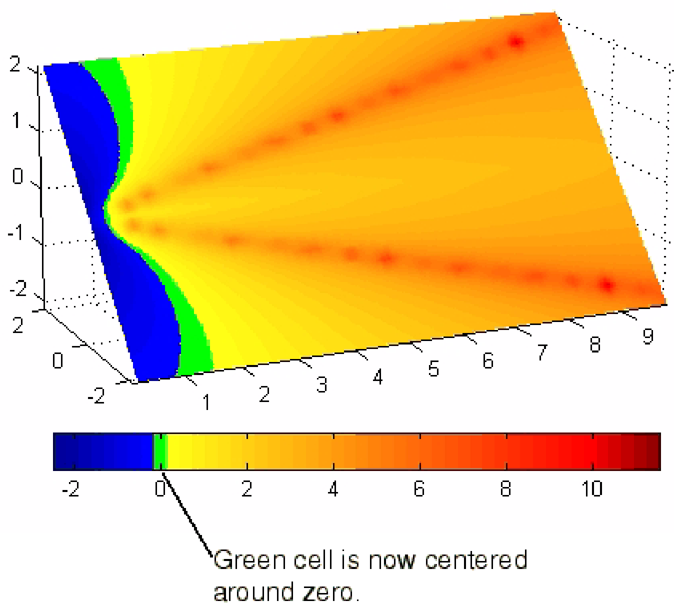
5 Before making further adjustments to the colormap, we need to move the green cell so that it is centered around zero. Use the colorbar to locate the green cell.



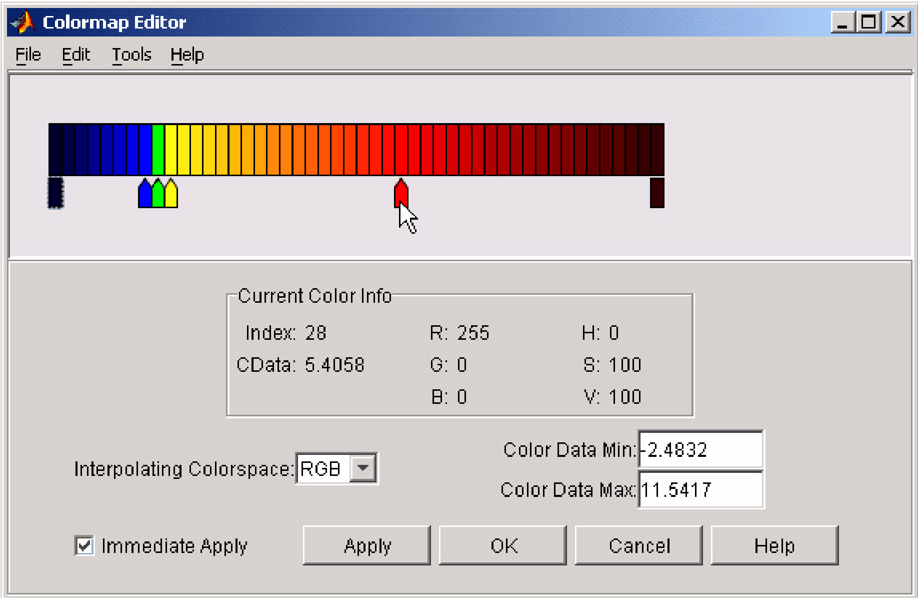
To recenter the green cell around zero, select the blue, green, and yellow node pointers (left-click blue, **Shift+click** yellow) and move them as a group using the left arrow key. Watch the colorbar in the figure window to see when the green color is centered around zero.



The slice plane now has the desired range of colors for negative, zero, and positive data.

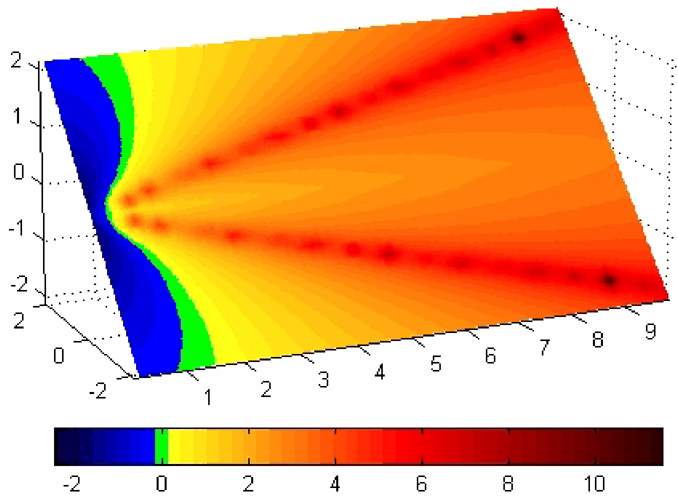


6 Increase the orange-red coloring in the slice by moving the red node pointer toward the yellow node.



7 Darken the endpoints to bring out more detail in the extremes of the data. Double-click the end nodes to display the color picker. Set the red endpoint to the RGB value [50 0 0] and set the blue endpoint to the RGB value [0 0 50].

The slice plane coloring now matches the example objectives.



Saving the Modified Colormap

You can save the modified colormap using the `colormap` function or the figure `Colormap` property.

After you have applied your changes, save the current figure colormap in a variable:

```
mycmap = get(fig,'Colormap'); % fig is figure
handle or use gcf
```

To use this colormap in another figure, set that figure's `Colormap` property:

```
set(new_fig,'Colormap',mycmap)
```

To save your modified colormap in a MAT-file, use the `save` command to save the `mycmap` workspace variable:

```
save('MyColormaps','mycmap')
```

To use your saved colormap in another MATLAB session, `load` the variable into the workspace and assign the colormap to the figure:

```
load('MyColormaps','mycmap')
set(fig,'Colormap',mycmap)
```

See Also

`colormap` | `get` | `load` | `propertyeditor` | `save` | `set`

How To

- "Colormaps"

Purpose

Color specification

Description

`ColorSpec` is not a function; it refers to the three ways in which you specify color for MATLAB graphics:

- RGB triple
- Short name
- Long name

The short names and long names are MATLAB strings that specify one of eight predefined colors. The RGB triple is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color; the intensities must be in the range [0 1]. The following table lists the predefined colors and their RGB equivalents.

RGB Value	Short Name	Long Name
`[1 1 0]`	`y`	`yellow`
`[1 0 1]`	`m`	`magenta`
`[0 1 1]`	`c`	`cyan`
`[1 0 0]`	`r`	`red`
`[0 1 0]`	`g`	`green`
`[0 0 1]`	`b`	`blue`
`[1 1 1]`	`w`	`white`
`[0 0 0]`	`k`	`black`

Tips

The eight predefined colors and any colors you specify as RGB values are not part of a figure's colormap, nor are they affected by changes to the figure's colormap. They are referred to as *fixed* colors, as opposed to *colormap* colors.

Some high-level functions (for example, `scatter`) accept a colorspec as an input argument and use it to set the `CData` of graphic objects they create. When using such functions, take care not to specify a colorspec

in a property/value pair that sets CData; values for CData are always n-length vectors or n-by-3 matrices, where n is the length of XData and YData, never strings.

Examples

To change the background color of a figure to green, specify the color with a short name, a long name, or an RGB triple. These statements generate equivalent results:

```
whitebg('g')
whitebg('green')
whitebg([0 1 0]);
```

You can use ColorSpec anywhere you need to define a color. For example, this statement changes the figure background color to pink:

```
set(gcf,'Color',[1,0.4,0.6])
```

See Also

bar | bar3 | colordef | colormap | fill | fill3 | whitebg

Purpose Sparse column permutation based on nonzero count

Syntax

```
j = colperm(S)
```

Description `j = colperm(S)` generates a permutation vector `j` such that the columns of `S(:,j)` are ordered according to increasing count of nonzero entries. This is sometimes useful as a preordering for LU factorization; in this case use `lu(S(:,j))`.

If `S` is symmetric, then `j = colperm(S)` generates a permutation `j` so that both the rows and columns of `S(j,j)` are ordered according to increasing count of nonzero entries. If `S` is positive definite, this is sometimes useful as a preordering for Cholesky factorization; in this case use `chol(S(j,j))`.

Algorithms The algorithm involves a sort on the counts of nonzeros in each column.

Examples The `n`-by-`n` *arrowhead* matrix

```
A = [ones(1,n); ones(n-1,1) speye(n-1,n-1)]
```

has a full first row and column. Its LU factorization, `lu(A)`, is almost completely full. The statement

```
j = colperm(A)
```

returns `j = [2:n 1]`. So `A(j,j)` sends the full row and column to the bottom and the rear, and `lu(A(j,j))` has the same nonzero structure as `A` itself.

On the other hand, the Bucky ball example,

```
B = bucky
```

has exactly three nonzero elements in each row and column, so `j = colperm(B)` is the identity permutation and is no help at all for reducing fill-in with subsequent factorizations.

See Also

chol | colamd | lu | spparms | symamd | symrcm

Purpose

Convenience function for static .NET System.Delegate Combine method

Syntax

```
result = Combine(delegateA,delegateB)
```

Description

`result = Combine(delegateA,delegateB)` combines two delegates into a new delegate.

Input Arguments

delegateA

.NET `System.Delegate` object. The first delegate in the new delegate.

delegateB

.NET `System.Delegate` object. The last delegate in the new delegate.

Output Arguments

result

.NET `System.Delegate` object. A new delegate that delegates to the input delegate `delegateA`, then `delegateB`

Alternatives

Use the static `Combine` method of the `System.Delegate` class.

See Also

`Remove` | `RemoveAll`

How To

- "Combine and Remove .NET Delegates"

Related Links

- MSDN System.Delegate.Combine Method reference page

comet

Purpose

2-D comet plot

Syntax

```
comet(y)
comet(x,y)
comet(x,y,p)
comet(axes_handle,...)
```

Description

`comet(y)` displays a comet graph of the vector `y`. A comet graph is an animated graph in which a circle (the comet *head*) traces the data points on the screen. The comet *body* is a trailing segment that follows the head. The *tail* is a solid line that traces the entire function.

`comet(x,y)` displays a comet graph of vector `y` versus vector `x`.

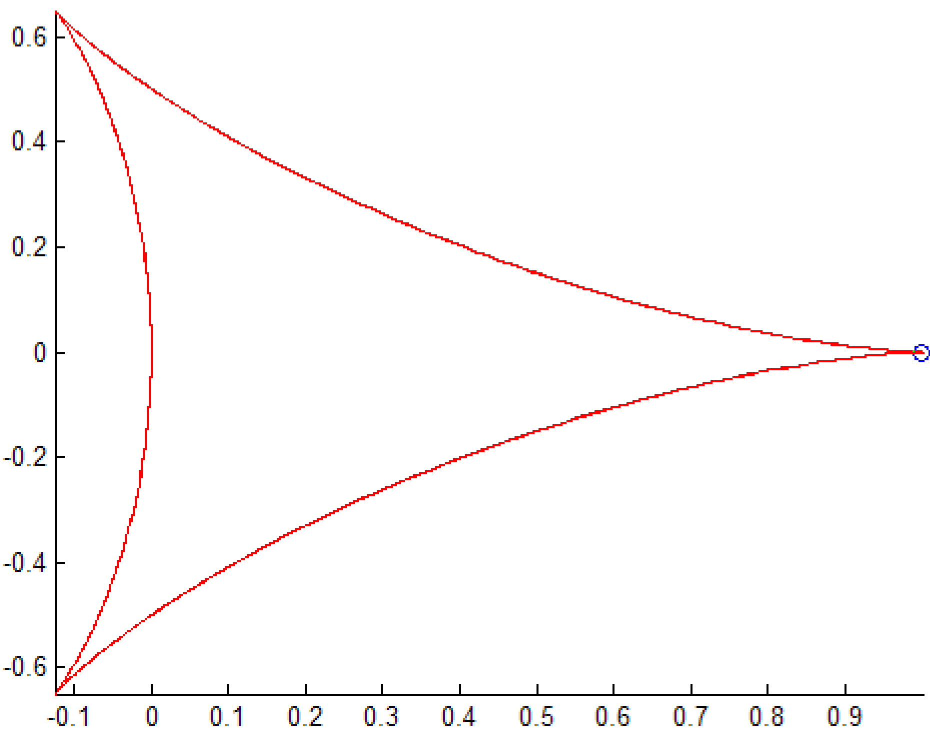
`comet(x,y,p)` specifies a comet body of length `p*length(y)`. `p` defaults to `0.1`.

`comet(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

Examples

Create a simple comet graph:

```
t = 0:.01:2*pi;
x = cos(2*t).*(cos(t).^2);
y = sin(2*t).*(sin(t).^2);
comet(x,y);
```

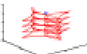


See Also

`comet3`

Purpose

3-D comet plot



Syntax

```
comet3(z)
comet3(x,y,z)
comet3(x,y,z,p)
comet3(axes_handle,...)
```

Description

A comet plot is an animated graph in which a circle (the comet *head*) traces the data points on the screen. The comet *body* is a trailing segment that follows the head. The *tail* is a solid line that traces the entire function.

`comet3(z)` displays a 3-D comet graph of the vector `z`.

`comet3(x,y,z)` displays a comet graph of the curve through the points `[x(i),y(i),z(i)]`.

`comet3(x,y,z,p)` specifies a comet body of length `p*length(y)`. `p` must be between `0` and `1`.

`comet3(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

Tips

The trace left by `comet3` is created by using an `EraseMode` of `none`, which means you cannot print the graph (you get only the comet head), and it disappears if you cause a redraw (for example, by resizing the window).

Examples

Create a 3-D comet graph.

```
t = -10*pi:pi/250:10*pi;
comet3((cos(2*t).^2).*sin(t),(sin(2*t).^2).*cos(t),t);
```

See Also

`comet`

commandhistory

Purpose

Open Command History window, or select it if already open

Syntax

```
commandhistory
```

Description

`commandhistory` opens the MATLAB Command History window when it is closed, and selects the Command History window when it is open. The Command History window presents a log of the statements most recently run in the Command Window.

See Also

`diary` | `prefdir` | `startup`

How To

- "Command History"

Purpose

Open Command Window, or select it if already open

Syntax

```
commandwindow
```

Description

`commandwindow` opens the MATLAB Command Window when it is closed, and selects the Command Window when it is open.

Tips

To determine the number of columns and rows that display in the Command Window, given its current size, use

```
get(0,'CommandWindowSize')
```

The number of columns is based on the width of the Command Window. With the matrix display width preference set to 80 columns, the number of columns is always 80.

See Also

`commandhistory` | `input` | `inputdlg`

How To

- "Optimize Desktop Layout for Limited Screen Space"
- "Set Command Window Preferences"

Purpose

Companion matrix

Syntax

```
A = compan(u)
```

Description

`A = compan(u)` returns the corresponding companion matrix whose first row is `-u(2:n)/u(1)`, where `u` is a vector of polynomial coefficients. The eigenvalues of `compan(u)` are the roots of the polynomial.

Examples

The polynomial $(x-1)(x-2)(x+3) = x^3 - 7x + 6$ has a companion matrix given by

```
u = [1  0  -7  6]
A = compan(u)
A =
     0     7    -6
     1     0     0
     0     1     0
```

The eigenvalues are the polynomial roots:

```
eig(compan(u))

ans =
    -3.0000
     2.0000
     1.0000
```

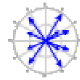
This is also `roots(u)`.

See Also

`eig` | `poly` | `polyval` | `roots`

Purpose

Plot arrows emanating from origin



Syntax

```
compass(U,V)
compass(Z)
compass(...,LineSpec)
compass(axes_handle,...)
h = compass(...)
```

Description

A compass graph displays the vectors with components `(U,V)` as arrows emanating from the origin. `U`, `V`, and `Z` are in Cartesian coordinates and plotted on a circular grid.

`compass(U,V)` displays a compass graph having *n* arrows, where *n* is the number of elements in `U` or `V`. The location of the base of each arrow is the origin. The location of the tip of each arrow is a point relative to the base and determined by `[U(i),V(i)]`.

`compass(Z)` displays a compass graph having *n* arrows, where *n* is the number of elements in `Z`. The location of the base of each arrow is the origin. The location of the tip of each arrow is relative to the base as determined by the real and imaginary components of `Z`. This syntax is equivalent to `compass(real(Z),imag(Z))`.

`compass(...,LineSpec)` draws a compass graph using the line type, marker symbol, and color specified by `LineSpec`.

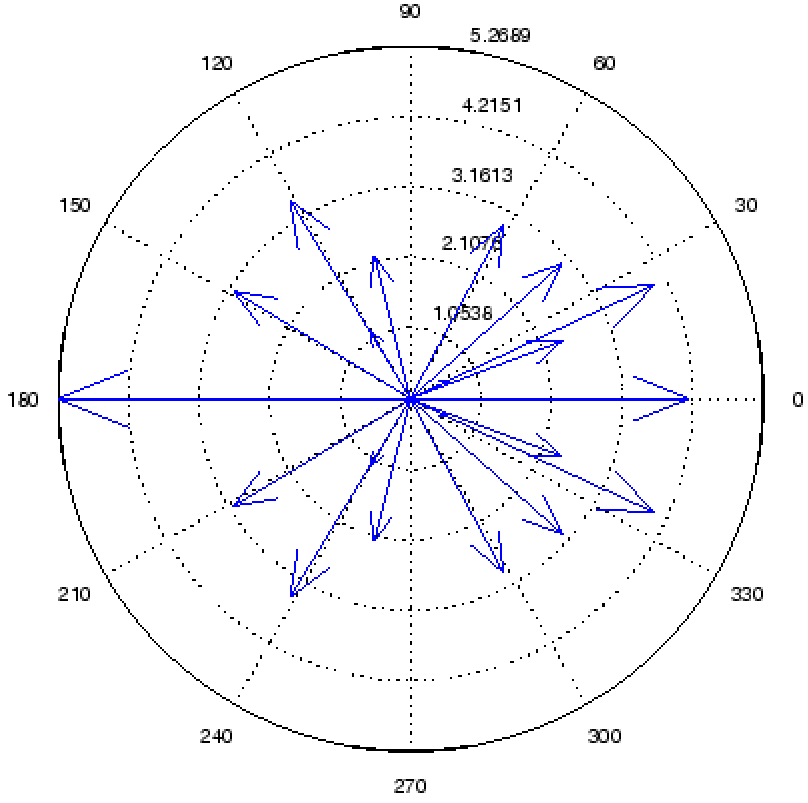
`compass(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = compass(...)` returns handles to line objects.

Examples

Draw a compass graph of the eigenvalues of a matrix.

```
Z = eig(randn(20,20));
compass(Z)
```



See Also

`feather` | `polar` | `LineSpec` | `quiver` | `rose`

How To

- "Compass Plots"

complex

Purpose

Construct complex data from real and imaginary components

Syntax

```
c = complex(a,b)
```

Description

`c = complex(a,b)` creates a complex output, `c`, from the two real inputs.

```
c = a + bi
```

The output is the same size as the inputs, which must be scalars or equally sized vectors, matrices, or multi-dimensional arrays.

Note If `b` is all zeros, `c` is complex and the value of all its imaginary components is `0`. In contrast, the result of the addition `a+0i` returns a strictly real result.

The following describes when `a` and `b` can have different data types, and the resulting data type of the output `c`:

- If either of `a` or `b` has type `single`, `c` has type `single`.
- If either of `a` or `b` has an integer data type, the other must have the same integer data type or type scalar `double`, and `c` has the same integer data type.

`c = complex(a)` for real `a` returns the complex result `c` with real part `a` and `0` as the value of all imaginary components. Even though the value of all imaginary components is `0`, `c` is complex and `isreal(c)` returns false.

The `complex` function provides a useful substitute for expressions such as

```
a + i*b   or   a + j*b
```

in cases when the names "i" and "j" may be used for other variables (and do not equal $\sqrt{-1}$), when a and b are not single or double, or when b is all zero.

Examples

Create complex uint8 vector from two real uint8 vectors.

```
a = uint8([1;2;3;4])
b = uint8([2;2;7;7])
c = complex(a,b)
c =
   1.0000 + 2.0000i
   2.0000 + 2.0000i
   3.0000 + 7.0000i
   4.0000 + 7.0000i
```

See Also

abs | angle | conj | i | imag | isreal | j | real

Tiff.computeStrip

Purpose

Index number of strip containing specified coordinate

Syntax

```
stripNumber = tiffobj.computeStrip(row)
stripNumber = tiffobj.computeStrip(row, plane)
```

Description

`stripNumber = tiffobj.computeStrip(row)` returns the index number of the strip containing the given row. The value of `row` must be one-based.

`stripNumber = tiffobj.computeStrip(row, plane)` returns the index number of the strip containing the given row in the specified plane, if the value of the `PlanarConfiguration` tag is `Tiff.PlanarConfiguration.Separate`.. The values of row and plane must be one-based.

`computeStrip` clamps out-of-range coordinate values to the bounds of the image.

Examples

Open a `Tiff` object and get the index number of the strip containing the middle row. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path:

```
t = Tiff('myfile.tif','r');
% Get the number of rows in the image.
numRows = t.getTag('ImageLength');
% Get the number of the strip containing the middle row
stripNum = t.computeStrip(numRows/2);
```

References

This method corresponds to the `TIFFComputeStrip` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff.computeTile`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

Purpose

Index number of tile containing specified coordinates

Syntax

```
tileNumber = tiffobj.computeTile([row col])
tileNumber = tiffobj.computeTile([row col], plane)
```

Description

`tileNumber = tiffobj.computeTile([row col])` returns the index number of the tile containing the row and column pixel coordinates. The row and column coordinate values are one-based.

`tileNumber = tiffobj.computeTile([row col], plane)` returns the index number of the tile containing the row and column pixel coordinates in the specified plane, if the value of the `PlanarConfiguration` tag is `Tiff.PlanarConfiguration.Separate`. The row, column, and plane coordinate values are one-based.

`computeTile` clamps out-of-range coordinate values to the bounds of the image.

Examples

Open a `Tiff` object and get the index number of the tile containing the last pixel. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path.

```
t = Tiff('myfile.tif','r');
% Get the dimensions of the image to calculate coordinates.
numRows = t.getTag('ImageLength');
numCols = t.getTag('ImageWidth');
% Get the ID number of the tile containing the coordinates.
tileNum = t.computeTile([numRows numCols]);
```

References

This method corresponds to the `TIFFComputeTile` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTiff - TIFF Library and Utilities`.

See Also

`Tiff.computeStrip`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"

- “Reading Image Data and Metadata from TIFF Files”

Purpose

Information about computer on which MATLAB software is running

Syntax

```
str = computer
archstr = computer('arch')
[str,maxsize] = computer
[str,maxsize,endian] = computer
```

Description

`str = computer` returns the string `str` with the computer type on which MATLAB is running.

`archstr = computer('arch')` returns the string `archstr` which is used by the `mex` command `-arch` switch.

`[str,maxsize] = computer` returns the integer `maxsize`, the maximum number of elements allowed in an array with this version of MATLAB.

`[str,maxsize,endian] = computer` returns either 'L' for little-endian byte ordering or 'B' for big-endian byte ordering.

Platform	Word Size	str	archstr	maxsize	endian	ispc	isunix	ismac
Microsoft Windows	32-bit	PCWIN	win32	2^31 - 1	L	1	0	0
	64-bit	PCWIN64	win64	2^48 - 1	L	1	0	0
Linux	64-bit	GLNXA64	glnxa64	2^48 - 1	L	0	1	0
Apple Macintosh	64-bit	MACI64	maci64	2^48 - 1	L	0	1	1

Tips

In some cases, both 32-bit and 64-bit versions of MATLAB can run on the same platform. In this case, the value returned by `computer` reflects which of these is running. For example, if you run a 32-bit version of MATLAB on a Windows x64 platform, `computer` returns `PCWIN`, indicating that the 32-bit version is running. You can get this information and the value of `archstr` from the **Help** menu, as described

in “Information About your Installation” in the Desktop Tools and Development Environment documentation.

See Also

`getenv` | `setenv` | `ispc` | `isunix` | `ismac`

Purpose

Condition number with respect to inversion

Syntax

```
c = cond(X)
c = cond(X,p)
```

Description

The *condition number* of a matrix measures the sensitivity of the solution of a system of linear equations to errors in the data. It gives an indication of the accuracy of the results from matrix inversion and the linear equation solution. Values of `cond(X)` and `cond(X,p)` near 1 indicate a well-conditioned matrix.

`c = cond(X)` returns the 2-norm condition number, the ratio of the largest singular value of `X` to the smallest.

`c = cond(X,p)` returns the matrix condition number in `p`-norm:

```
norm(X,p) * norm(inv(X),p)
```

If p is...	Then cond(X,p) returns the...
1	1-norm condition number
2	2-norm condition number
'fro'	Frobenius norm condition number
inf	Infinity norm condition number

Algorithms

The algorithm for `cond` (when `p = 2`) uses the singular value decomposition, `svd`. When the input matrix is sparse, `cond` ignores any specified `p` value and calls `condest`.

See Also

`condeig` | `condest` | `norm` | `normest` | `rank` | `rcond` | `svd`

Purpose

Condition number with respect to eigenvalues

Syntax

```
c = condeig(A)
[V,D,s] = condeig(A)
```

Description

`c = condeig(A)` returns a vector of condition numbers for the eigenvalues of `A`. These condition numbers are the reciprocals of the cosines of the angles between the left and right eigenvectors.

`[V,D,s] = condeig(A)` is equivalent to

```
[V,D] = eig(A);
s = condeig(A);
```

Large condition numbers imply that `A` is near a matrix with multiple eigenvalues.

See Also

`balance` | `cond` | `eig`

Purpose

1-norm condition number estimate

Syntax

```
c = condest(A)
c = condest(A,t)
[c,v] = condest(A)
```

Description

`c = condest(A)` computes a lower bound `c` for the 1-norm condition number of a square matrix `A`.

`c = condest(A,t)` changes `t`, a positive integer parameter equal to the number of columns in an underlying iteration matrix. Increasing the number of columns usually gives a better condition estimate but increases the cost. The default is `t = 2`, which almost always gives an estimate correct to within a factor 2.

`[c,v] = condest(A)` also computes a vector `v` which is an approximate null vector if `c` is large. `v` satisfies `norm(A*v,1) = norm(A,1)*norm(v,1)/c`.

Note `condest` invokes `rand`. If repeatable results are required then use `rng` to set the random number generator to its startup settings before using `condest`.

```
rng('default')
```

Tips

This function is particularly useful for sparse matrices.

Algorithms

`condest` is based on the 1-norm condition estimator of Hager [1] and a block-oriented generalization of Hager's estimator given by Higham and Tisseur [2]. The heart of the algorithm involves an iterative search to estimate $\left\|A^{-1}\right\|_1$ without computing A^{-1}. This is posed as the convex but nondifferentiable optimization problem $\max\left\|A^{-1}\mathbf{x}\right\|_1$ subject to $\|\mathbf{x}\|_1 = 1$

References

[1] William W. Hager, "Condition Estimates," *SIAM J. Sci. Stat. Comput. 5*, 1984, 311-316, 1984.

[2] Nicholas J. Higham and Françoise Tisseur, "A Block Algorithm for Matrix 1-Norm Estimation with an Application to 1-Norm Pseudospectra, "*SIAM J. Matrix Anal. Appl.*, Vol. 21, 1185-1201, 2000.

See Also

`cond` | `norm` | `normest`

Purpose

Plot velocity vectors as cones in 3-D vector field

Syntax

```
coneplot(X,Y,Z,U,V,W,Cx,Cy,Cz)
coneplot(U,V,W,Cx,Cy,Cz)
coneplot(...,s)
coneplot(...,color)
coneplot(...,'quiver')
coneplot(...,'method')
coneplot(X,Y,Z,U,V,W,'nointerp')
coneplot(axes_handle,...)
h = coneplot(...)
```

Description

`coneplot(X,Y,Z,U,V,W,Cx,Cy,Cz)` plots velocity vectors as cones pointing in the direction of the velocity vector and having a length proportional to the magnitude of the velocity vector. `X`, `Y`, `Z` define the coordinates for the vector field. `U`, `V`, `W` define the vector field. These arrays must be the same size, monotonic, and represent a Cartesian, axis-aligned grid (such as the data produced by `meshgrid`). `Cx`, `Cy`, `Cz` define the location of the cones in the vector field. The section "Specifying Starting Points for Stream Plots" in Visualization Techniques provides more information on defining starting points.

`coneplot(U,V,W,Cx,Cy,Cz)` (omitting the `X`, `Y`, and `Z` arguments) assumes `[X,Y,Z] = meshgrid(1:n,1:m,1:p)`, where `[m,n,p]= size(U)`.

`coneplot(...,s)` automatically scales the cones to fit the graph and then stretches them by the scale factor `s`. If you do not specify a value for `s`, `coneplot` uses a value of 1. Use `s = 0` to plot the cones without automatic scaling.

`coneplot(...,color)` interpolates the array `color` onto the vector field and then colors the cones according to the interpolated values. The size of the `color` array must be the same size as the `U`, `V`, `W` arrays. This option works only with cones (that is, not with the `quiver` option).

`coneplot(...,'quiver')` draws arrows instead of cones (see `quiver3` for an illustration of a quiver plot).

`coneplot(...,'method')` specifies the interpolation method to use. *`method`* can be `linear`, `cubic`, or `nearest`. `linear` is the default. (See `interp3` for a discussion of these interpolation methods.)

`coneplot(X,Y,Z,U,V,W,'nointerp')` does not interpolate the positions of the cones into the volume. The cones are drawn at positions defined by `X`, `Y`, `Z` and are oriented according to `U`, `V`, `W`. Arrays `X`, `Y`, `Z`, `U`, `V`, `W` must all be the same size.

`coneplot(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = coneplot(...)` returns the handle to the `patch` object used to draw the cones. You can use the `set` command to change the properties of the cones.

`coneplot` automatically scales the cones to fit the graph, while keeping them in proportion to the respective velocity vectors.

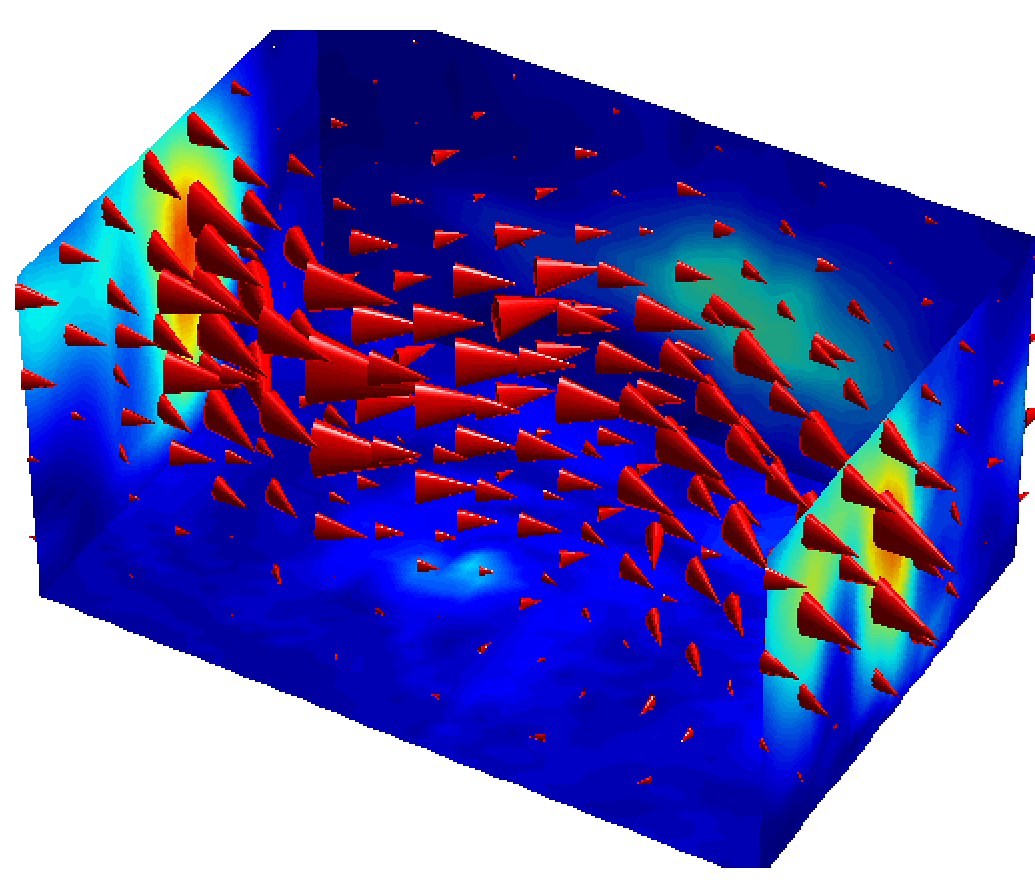
Examples

Plot the velocity vector cones for vector volume data representing the motion of air through a rectangular region of space:

```
% Load the data. The winds data set contains six 3-D arrays: u, v, and w specify
% the vector components at each of the coordinates specified in x, y, and z. The
% coordinates define a lattice grid structure where the data is sampled within the
% volume.
load wind

% Now establish the range of the data to place the slice planes and to specify
% where you want the cone plots (min, max):
xmin = min(x(:));
xmax = max(x(:));
ymin = min(y(:));
ymax = max(y(:));
zmin = min(z(:));

% Use daspect to set the data aspect ratio of the axes before calling coneplot.
daspect([2,2,1])
```

```
% Decide where in data space you want to plot cones. This example selects the
% full range of x and y in eight steps and the range 3 to 15 in four steps in z
% using linspace and meshgrid.
xrange = linspace(xmin,xmax,8);
yrange = linspace(ymin,ymax,8);
zrange = 3:4:15;
[cx cy cz] = meshgrid(xrange,yrange,zrange);

% Draw the cones, setting the scale factor to 5 to make the cones larger than
% the default size:
hcones = coneplot(x,y,z,u,v,w,cx,cy,cz,5);
% Set the coloring of each cone using FaceColor and EdgeColor:
set(hcones,'FaceColor','red','EdgeColor','none')

% Calculate the magnitude of the vector field (which represents wind speed) to
% generate scalar data for the slice command:
hold on
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
% Create slice planes along the x-axis at xmin and xmax, along the y-axis at
% ymax, and along the z-axis at zmin:
hsurfaces = slice(x,y,z,wind_speed,[xmin,xmax],ymax,zmin);
% Specify interpolated face color so the slice coloring indicates wind speed,
% and do not draw edges (hold, slice, FaceColor, EdgeColor):
set(hsurfaces,'FaceColor','interp','EdgeColor','none')
hold off

% Use the axis command to set the axis limits equal to the range of the data.
axis tight;
% Orient the view to azimuth = 30 and elevation = 40. (rotate3d is a useful
% command for selecting the best view.)
view(30,40);axis off
% Select perspective projection to provide a more realistic looking volume
% using camproj:
camproj perspective;
% Zoom in on the scene a little to make the plot as large as possible using camzoom:
camzoom(1.5)
```

```
% The light source affects both the slice planes (surfaces) and the cone plots
% (patches). However, you can set the lighting characteristics of each independently:

% Add a light source to the right of the camera and use Phong lighting to give the
% cones and slice planes a smooth, three-dimensional appearance using camlight and lighting:
camlight right; lighting phong

% Increase the value of the AmbientStrength property for each slice plane to improve
% the visibility of the dark blue colors:
set(hsurfaces,'AmbientStrength',.6)
% Increase the value of the DiffuseStrength property of the cones to brighten particularly
% those cones not showing specular reflections:
set(hcones,'DiffuseStrength',.8)
```



See Also

`isosurface` | `patch` | `reducevolume` | `smooth3` | `streamline` | `stream2` | `stream3` | `subvolume`

Tutorials

- “Overview of Volume Visualization”

conj

Purpose

Complex conjugate

Syntax

```
ZC = conj(Z)
```

Description

`ZC = conj(Z)` returns the complex conjugate of the elements of `Z`.

Algorithms

If `Z` is a complex array:

```
conj(Z) = real(Z) - i*imag(Z)
```

See Also

`i` | `j` | `imag` | `real`

Purpose

Pass control to next iteration of `for` or `while` loop

Syntax

```
continue
```

Description

`continue` temporarily interrupts the execution of a program loop, skipping any remaining statements in the body of the loop for the current pass. The `continue` statement does not cause an immediate exit from the loop as a `break` or `return` statement would do, but instead continues within the loop for as long as the stated `for` or `while` condition holds true.

A `continue` statement in a *nested* loop behaves in the same manner. Execution resumes at the `for` or `while` statement of the loop in which the `continue` statement was encountered, and reenters the loop if the stated condition evaluates to `true`.

Examples

Count the number of lines of code in the file `magic.m`, skipping all blank lines and comments:

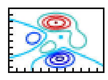
```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) || strncmp(line,'%',1) || ~ischar(line)
        continue
    end
    count = count + 1;
end
fprintf('%d lines\n',count);
fclose(fid);
```

See Also

`for` | `while` | `end` | `break` | `return`

Purpose

Contour plot of matrix



Syntax

```
contour(Z)
contour(Z,n)
contour(Z,v)
contour(X,Y,Z)
contour(X,Y,Z,n)
contour(X,Y,Z,v)
contour(...,LineSpec)
contour(axes_handle,...)
[C,h] = contour(...)
```

Description

A contour plot displays isolines of matrix `Z`. Label the contour lines using `clabel`.

`contour(Z)` draws a contour plot of matrix `Z`, where `Z` is interpreted as heights with respect to the *x*-*y* plane. `Z` must be at least a 2-by-2 matrix that contains at least two different values. The number of contour lines and the values of the contour lines are chosen automatically based on the minimum and maximum values of `Z`. The ranges of the *x*- and *y*-axis are `[1:n]` and `[1:m]`, where `[m,n] = size(Z)`.

`contour(Z,n)` draws a contour plot of matrix `Z` with `n` contour levels where `n` is a scalar.

`contour(Z,v)` draws a contour plot of matrix `Z` with contour lines at the data values specified in the monotonically increasing vector `v`. The number of contour levels is equal to `length(v)`. To draw a single contour of level `i`, use `contour(Z,[i i])`. Specifying the vector v sets the `LevelListMode` to manual to allow user control over contour levels. See `contourgroup properties` for more information.

`contour(X,Y,Z)`, `contour(X,Y,Z,n)`, and `contour(X,Y,Z,v)` draw contour plots of `Z` using `X` and `Y` to determine the *x*- and *y*-axis limits.

When X and Y are matrices, they must be the same size as Z and must be monotonically increasing.

contour(...,LineSpec) draws the contours using the line type and color specified by LineSpec. contour ignores marker symbols.

contour(axes_handle,...) plots into axes axes_handle instead of gca.

[C,h] = contour(...) returns a contour matrix, C, that contains the data that defines the contour lines, and a handle, h, to a contourgroup object. The clabel function uses contour matrix C to label the contour lines. ContourMatrix is also a read-only contourgroup property that you can obtain from the returned handle.

Use contourgroup object properties to control the contour plot appearance.

If X or Y is irregularly spaced, contour calculates contours using a regularly spaced contour grid, and then transforms the data to X or Y.

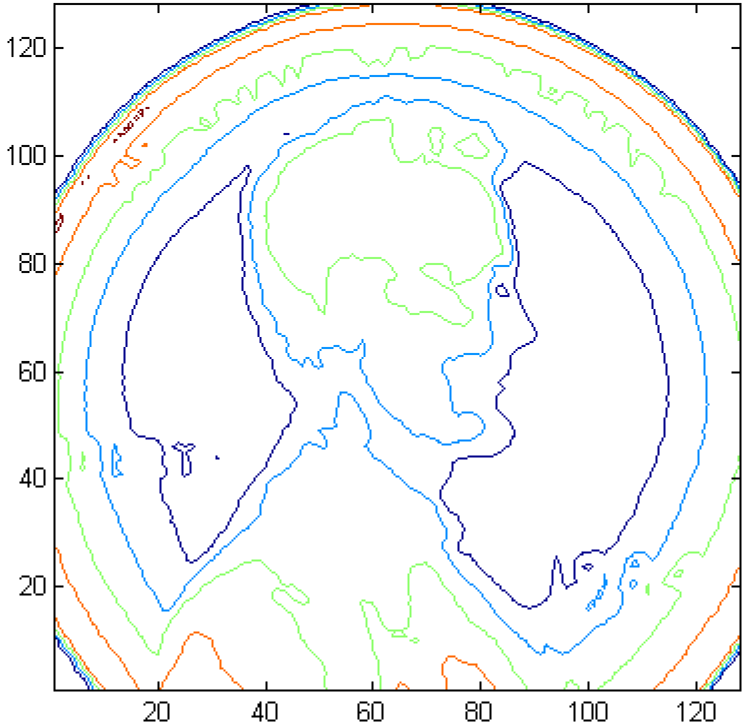
Examples

Contour Graph of a Data Set

Load the penny data set, which is a 128x128 grid of the surface relief of a U.S. penny. It generates a variable *P*. To contour this data, pass the matrix into the contour function. You have to flip the graph using flipud to see the actual contour of the image on the penny:

```
load penny;
figure;
contour(flipud(P));
axis square;
```

The figure output is as:

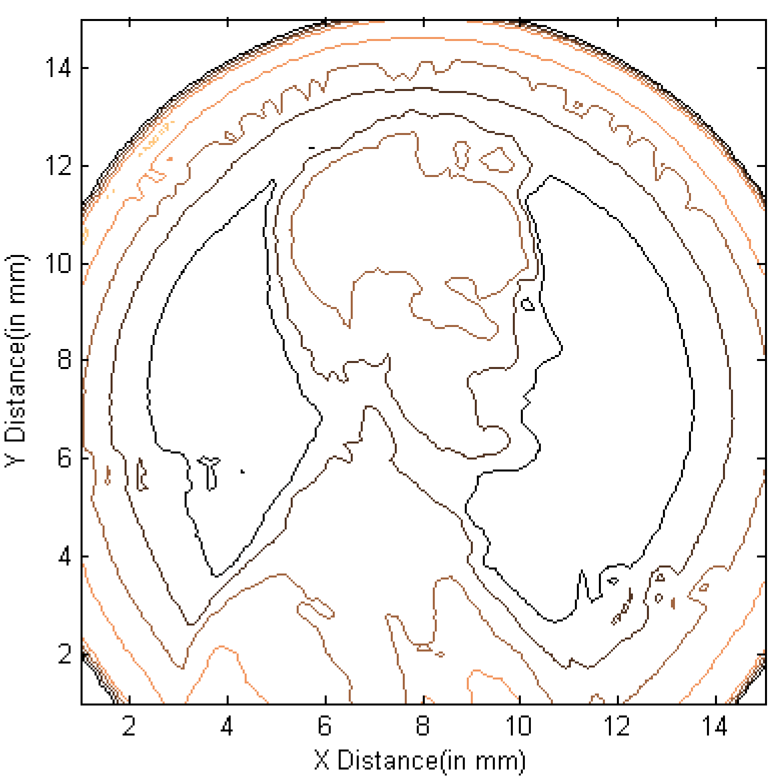


Contour Graph of a Matrix

Consider plotting the `penny` dataset with respect to the diameter distance. Assume the starting point to be `1mm` and ending at around `15mm`, we need 128 data points on both x and y axes because the dataset is a `128x128 double`. To generate 128 elements for both axes, we use `linspace` function here.

```
x = linspace(1,15,128);
y = linspace(1,15,128);
figure;
```

```
contour(x,y,flipud(P));
axis square;
colormap('Copper');
xlabel('X Distance(in mm)');
ylabel('Y Distance(in mm)');
```



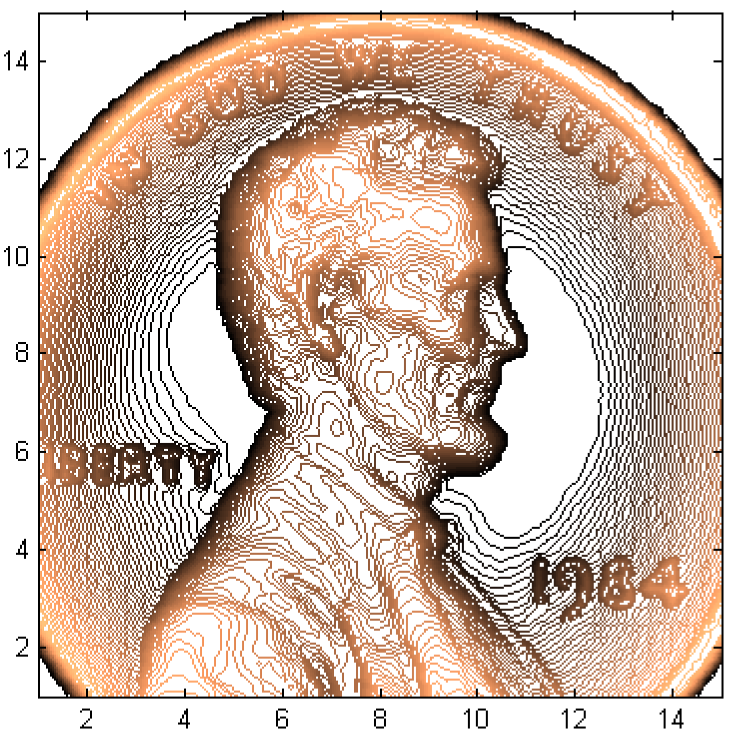
Contour Graph with Specified Number of Contour Lines

To plot a contour with more levels,

```
figure;
```

```
contour(x,y,flipud(P),50);
axis square;
colormap('Copper');
```

The figure looks like this.



Contour Graph of a Function

Create a contour plot of the function:

$z = xe^{(-x_2 - y_2)}$

over the range $-2 \le x \le 2$, $-2 \le y \le 3$.

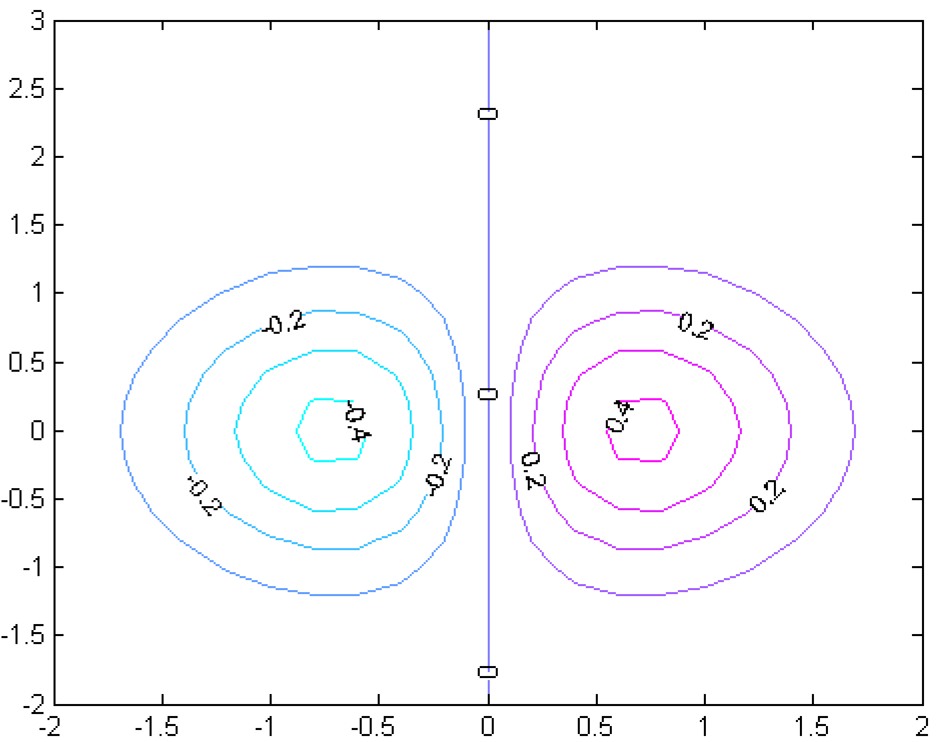
Evaluate the function to create matrix, Z. Use the meshgrid function to generate the values used to evaluate the function within the specified range:

```
[X,Y] = meshgrid(-2:.2:2,-2:.2:3);
Z = X.*exp(-X.^2-Y.^2);
```

Generate the contour plot of Z:

- Display contour labels by setting the ShowText property to on.
- Label every other contour line by setting the TextStep property to twice the contour interval (that is, two times the LevelStep property).
- Use a smoothly varying colormap.

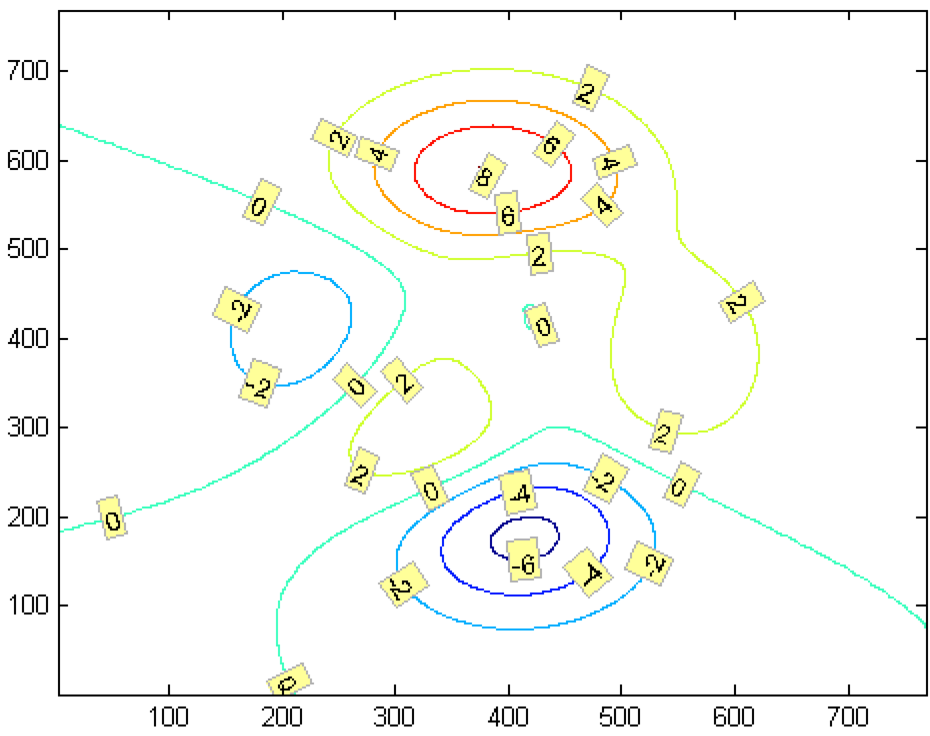
```
[C,h] = contour(X,Y,Z);
set(h,'ShowText','on','TextStep',get(h,'LevelStep')*2)
colormap cool
```



Smoothing Contour Data

Use `interp2` to smooth contour lines. Also set the contour label text `BackgroundColor` to a light yellow and the `EdgeColor` to light gray.

```
Z = peaks;
[C,h] = contour(interp2(Z,4));
text_handle = clabel(C,h);
set(text_handle,'BackgroundColor',[1 1 .6],...
    'Edgecolor',[.7 .7 .7])
```



For more examples using `contour`, see “Contour Plots”.

Data With Discontinuities

The `contour` function cannot determine if there are discontinuities in the input data. You can add `NaN`s to the data to prevent drawing the contour lines in those regions. For example, add `NaN` to the `y` data where `y == 0` and `x < 0`:

```
[x,y] = meshgrid(-2.05:.1:2.05,[-2.05:.1:-0.05,0,0.05:.1:2.05]);
y(y==0 & x<0) = NaN; % Explicitly add NaNs to data
```

```
contour(x,y,atan2(y,x))
```

See Also

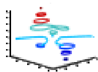
clabel | contourf | contour3 | contourc | quiver | contourgroup properties | text properties

How To

- "Contour Plots"

Purpose

3-D contour plot



Syntax

```
contour3(Z)
contour3(Z,n)
contour3(Z,v)
contour3(X,Y,Z)
contour3(X,Y,Z,n)
contour3(X,Y,Z,v)
contour3(...,LineSpec)
contour3(axes_handle,...)
[C,h] = contour3(...)
```

Description

`contour3` creates a 3-D contour plot of a surface defined on a rectangular grid.

`contour3(Z)` draws a contour plot of matrix `Z` in a 3-D view. `Z` is interpreted as heights with respect to the *x*-*y* plane. `Z` must be at least a 2-by-2 matrix that contains at least two different values. The number of contour levels and the values of contour levels are chosen automatically based on the minimum and maximum values of `Z`. The ranges of the *x*- and *y*-axis are `[1:n]` and `[1:m]`, where `[m,n] = size(Z)`.

`contour3(Z,n)` draws a contour plot of matrix `Z` with `n` contour levels in a 3-D view.

`contour3(Z,v)` draws a contour plot of matrix `Z` with contour lines at the values specified in vector `v`. The number of contour levels is equal to `length(v)`. To draw a single contour of level `i`, use `contour(Z,[i i])`. Specifying the vector v sets the `LevelListMode` to manual to allow user control over contour levels. See `contourgroup properties` for more information.

`contour3(X,Y,Z)`, `contour3(X,Y,Z,n)`, and `contour3(X,Y,Z,v)` draw contour plots of `Z` using `X` and `Y` to determine the *x*- and *y*-axis limits. If `X` is a matrix, `X(1,:)` defines the *x*-axis. If `Y` is a matrix, `Y(:,1)` defines

the *y*-axis. When X and Y are matrices, they must be the same size as Z and must be monotonically increasing.

`contour3(...,LineSpec)` draws the contour lines using the line type and color specified by `LineSpec`. `contour3` ignores marker symbols.

`contour3(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`[C,h] = contour3(...)` returns a contour matrix, `C`, that contains the data that defines the contour lines, and a handle, `h`, to an array of handles to graphics objects. The `clabel` function uses contour matrix `C` to label the contour lines. The graphic objects that `contour3` creates are `patch` objects, or if you specify a `LineSpec` argument, `line` objects.

Tips

If `X` or `Y` is irregularly spaced, `contour3` calculates contours using a regularly spaced contour grid, and then transforms the data to `X` or `Y`.

If you do not specify `LineSpec`, the functions `colormap` and `caxis` control the color.

Label the contour lines using `clabel`.

`contour3(...)` works the same as `contour(...)` with these exceptions:

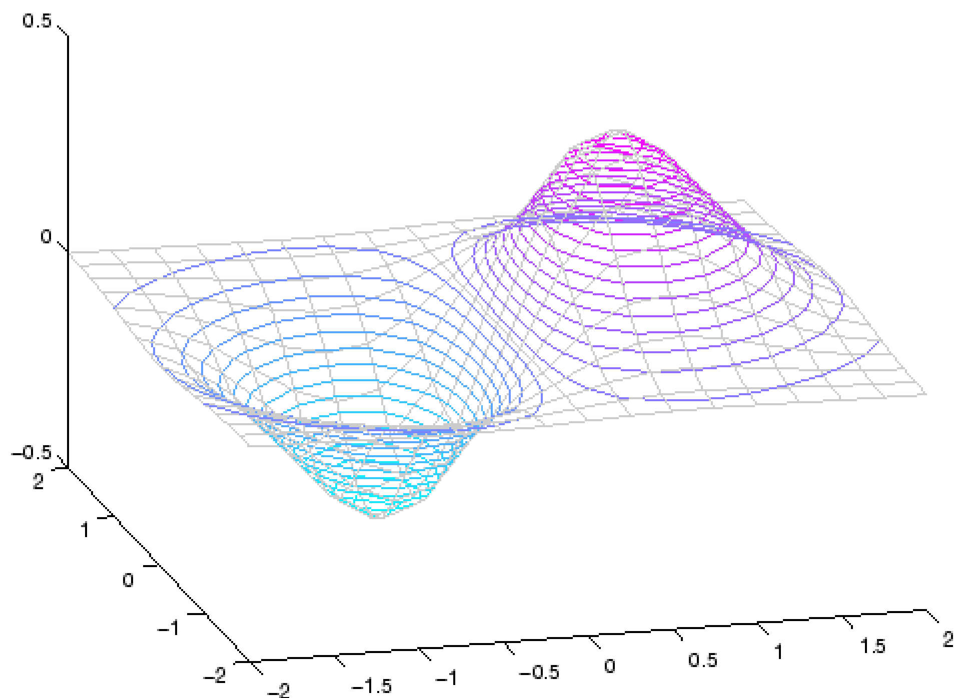
- The contours are drawn at their corresponding Z level.
- Multiple `patch` or `line` objects are created instead of a `contourgroup`.
- Calling `contour3` with trailing property-value pairs is not allowed.

Examples

Plot the three-dimensional contour of a function and superimpose a surface plot to enhance visualization of the function.

```
[X,Y] = meshgrid([-2:.25:2]);
Z = X.*exp(-X.^2-Y.^2);
contour3(X,Y,Z,30)
surface(X,Y,Z,'EdgeColor',[.8 .8 .8],'FaceColor','none')
grid off
view(-15,25)
```

```
colormap cool
```



For more examples using `contour3`, see “Contour Plots”.

See Also

`contour` | `contourc` | `contourf` | `meshc` | `meshgrid` | `surfc` | `contourgroup properties`

How To

- “Contour Plots”

contourc

Purpose

Low-level contour plot computation

Syntax

```
C = contourc(Z)
C = contourc(Z,n)
C = contourc(Z,v)
C = contourc(x,y,Z)
C = contourc(x,y,Z,n)
C = contourc(x,y,Z,v)
```

Description

`contourc` calculates the contour matrix `C` used by `contour`, `contour3`, and `contourf`. The values in `Z` determine the heights of the contour lines with respect to a plane. The contour calculations use a regularly spaced grid determined by the dimensions of `Z`.

`C = contourc(Z)` computes the contour matrix from data in matrix `Z`, where `Z` must be at least a 2-by-2 matrix. The contours are isolines in the units of `Z`. The number of contour lines and the corresponding values of the contour lines are chosen automatically.

`C = contourc(Z,n)` computes contours of matrix `Z` with `n` contour levels.

`C = contourc(Z,v)` computes contours of matrix `Z` with contour lines at the values specified in vector `v`. The length of `v` determines the number of contour levels. To compute a single contour of level `i`, use `contourc(Z,[i i])`.

`C = contourc(x,y,Z)`, `C = contourc(x,y,Z,n)`, and `C = contourc(x,y,Z,v)` compute contours of `Z` using vectors `x` and `y` to determine the *x*- and *y*-axis limits. `x` and `y` must be monotonically increasing.

Tips

`C` is a two-row matrix specifying all the contour lines. Each contour line defined in matrix `C` begins with a column that contains the value of the contour (specified by `v` and used by `clabel`), and the number of `(x,y)` vertices in the contour line. The remaining columns contain the data for the `(x,y)` pairs.

```
C = [value1 xdata(1) xdata(2) ... xdata(dim1) value2 xdata(1) xdata(2) ..
```

```
dim1 ydata(1) ydata(2) ... ydata(dim1) dim2 ydata(1) ydata(2) ... y
```

Specifying irregularly spaced `x` and `y` vectors is not the same as contouring irregularly spaced data. If `x` or `y` is irregularly spaced, `contourc` calculates contours using a regularly spaced contour grid, then transforms the data to `x` or `y`.

See Also

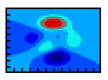
`clabel` | `contour` | `contour3` | `contourf`

How To

- "The Contouring Algorithm"

Purpose

Filled 2-D contour plot



Syntax

```
contourf(Z)
contourf(Z,n)
contourf(Z,v)
contourf(X,Y,Z)
contourf(X,Y,Z,n)
contourf(X,Y,Z,v)
contourf(...,LineSpec)
contourf(axes_handle,...)
contour(axes_handle,...)
[C,h] = contourf(...)
```

Description

A filled `contour` plot displays isolines calculated from matrix `Z` and fills the areas between the isolines using constant colors corresponding to the current figure's colormap.

`contourf(Z)` draws a filled contour plot of matrix `Z`, where `Z` is interpreted as heights with respect to the *x*-*y* plane. `Z` must be at least a 2-by-2 matrix that contains at least two different values. The number of contour lines and the values of the contour lines are chosen automatically based on the minimum and maximum values of `Z`. The ranges of the *x*- and *y*-axis are `[1:n]` and `[1:m]`, where `[m,n] = size(Z)`.

`contourf(Z,n)` draws a filled contour plot of matrix `Z` with `n` contour levels.

`contourf(Z,v)` draws a filled contour plot of matrix `Z` with contour lines at the data values specified in the monotonically increasing vector `v`. The number of contour levels is equal to `length(v)`. To draw a single contour of level `i`, use `contour(Z,[i i])`. Specifying the vector v sets the `LevelListMode` to manual to allow user control over contour levels. See `contourgroup properties` for more information.

`contourf(X,Y,Z)`, `contourf(X,Y,Z,n)`, and `contourf(X,Y,Z,v)` draw filled contour plots of `Z` using `X` and `Y` to determine the *x*- and *y*-axis limits. When `X` and `Y` are matrices, they must be the same size as `Z` and must be monotonically increasing.

`contourf(...,LineSpec)` draws the contour lines using the line type and color specified by `LineSpec`. `contourf` ignores marker symbols.

`contourf(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`contour(axes_handle,...)` plots into axes `gerkaxes_handle` instead of `gca`.

`[C,h] = contourf(...)` returns a contour matrix, `C`, that contains the data that defines the contour lines, and a handle, `h`, to a contourgroup object containing the filled contours. The `clabel` function uses contour matrix `C` to label the contour lines. `ContourMatrix` is also a read-only Contourgroup property that you can obtain from the returned handle.

Tips

Use `contourgroup` object properties to control the filled contour plot appearance.

Label the contour lines using `clabel`.

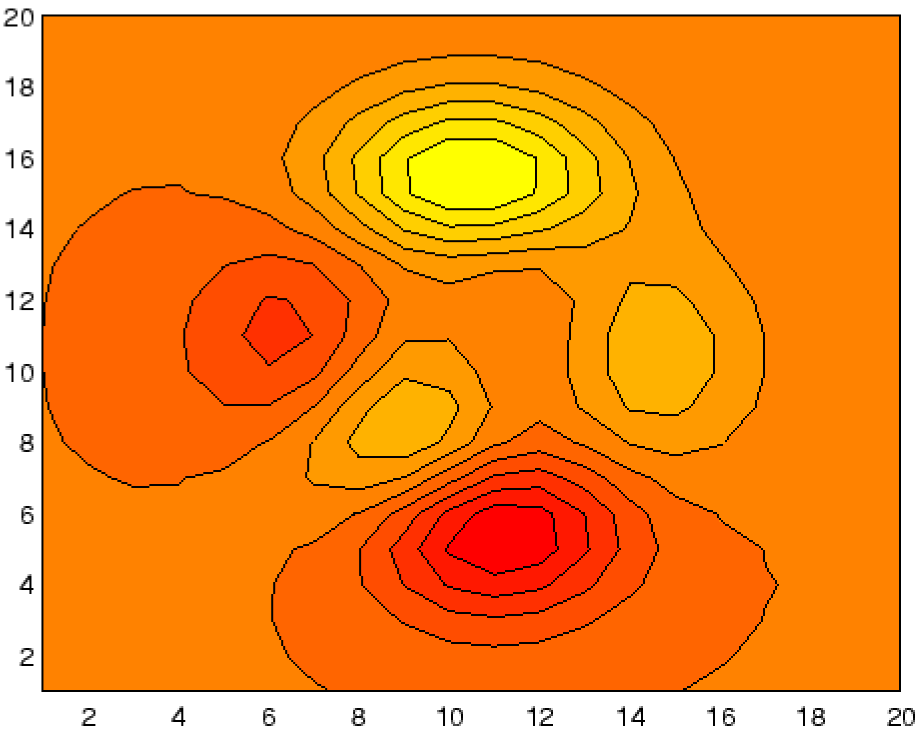
`NaN`s in the Z-data leave white holes with black borders in the contour plot.

If `X` or `Y` is irregularly spaced, `contourf` calculates contours using a regularly spaced contour grid, and then transforms the data to `X` or `Y`.

Examples

Create a filled contour plot of the `peaks` function with contour matrix and `contourgroup` object handle as output and `autumn` colormap.

```
[C,h] = contourf(peaks(20),10);
colormap autumn
```



For more examples using `contourf`, see "Contour Plots".

See Also

`clabel` | `contour` | `contour3` | `contourc` | `quiver` | `contourgroup properties`

How To

- "Contour Plots"

Purpose

Define contourgroup properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands or the Property Editor (`propertyeditor`).

Note that you cannot define default properties for contourgroup objects.

See "Plot Objects" for more information on contourgroup objects.

Contourgroup Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`Annotation`

`hg.Annotation` object (read-only)

Control the display of contourgroup objects in legends. Specifies whether this contourgroup object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the contourgroup object is displayed in a figure legend.

IconDisplayStyle Value	Purpose
`on`	Include the contourgroup object in a legend as one entry, but not its children objects
`off`	Do not include the contourgroup or its children in a legend (default)
`children`	Include only the children of the contourgroup as separate entries in the legend

Setting the IconDisplayStyle Property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the IconDisplayStyle Property

See "Controlling Legends" for more information and examples.

`BeingDeleted`

`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the

running callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`

string | function handle

Button press callback function. Executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be:

- A string that is a valid MATLAB expression
- The name of a MATLAB file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`Children`

array of graphics object handles

Children of the contourgroup object. An array containing the handles of all line objects parented to the contourgroup object (whether visible or not).

If a child object's `HandleVisibility` property is `callback` or `off`, its handle does not show up in this object's `Children` property. If you want the handle in the `Children` property, set the root `ShowHiddenHandles` property to `on`. For example:

```
set(0,'ShowHiddenHandles','on')
```

`Clipping`

`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs appear outside the axes plot box. This occurs if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`ContourMatrix`

`2`-by-`n` matrix (read-only)

Two-row matrix specifying all contour lines. Each contour line defined in the `ContourMatrix` begins with a column that contains the value of the contour (specified by the `LevelList` property and is used by `clabel`), and the number of `(x,y)` vertices in the contour line. The remaining columns contain the data for the `(x,y)` pairs.

For example:

```
C = [value1 xdata(1) xdata(2)...value2 xdata(1) xdata(2)...;
       dim1 ydata(1) ydata(2)...  dim2 ydata(1) ydata(2)...]
```

That is:

```
C = [C(1) C(2)...C(I)...C(N)]
```

where N is the number of contour levels, and:

```
C(i) = [ level(i) x(1) x(2)...x(  numel(i));
         numel(i) y(1) y(2)...y(  numel(i))];
```

For further information, see `contour` and The Contouring Algorithm.

`CreateFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback function executed during object creation. Defines a callback that executes when MATLAB creates an object. The default is an empty array.

You must specify the callback during the creation of the object. For example:

```
graphicfcn(y,'CreateFcn',@CallbackFcn)
```

where *`@CallbackFcn`* is a function handle that references the callback function and *`graphicfcn`* is the plotting function which creates this object.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DeleteFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback executed during object deletion. Executes when this object is deleted (for example, this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values. The default is an empty array.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`

string

String used by legend. The `legend` function uses the `DisplayName` property to label the contourgroup object in the legend. The default is an empty string.

- If you specify string arguments with the `legend` function, MATLAB set `DisplayName` to the corresponding string and uses that string for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, MATLAB set `DisplayName` to the edited string.

- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add a legend programmatically that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more information and examples.

`EraseMode`
`{normal} | none | xor | background`

Erase mode. Controls the technique MATLAB uses to draw and erase objects. Use alternative erase modes for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the object when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode` `none`, you cannot print it because MATLAB stores no information about its former location.
- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object's color depends on the color of whatever is beneath it on the display.
- `background` — Erase the object by redrawing it in the axes background `Color`, or the figure background `Color` if the axes `Color` property is `none`. This damages objects that are behind the erased object, but properly colors the erased object.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors (for example, performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`Fill`
`{off} | on`

Color spaces between contour lines.

- `on` — Default when using `contourf` or `ezcontourf`.
- `off` — Default when using `contour` or `ezcontour`.

By default, `contour` draws only the contour lines of the surface. If you set `Fill` to `on`, `contour` colors the regions in between the contour lines according to the Z-value of the region and changes the contour lines to black.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- on — Handles are always visible.
- callback — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- off — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes get, findobj, gca, gcf, gco, newplot, cla, clf, and close.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using callback or off, the object's handle does not appear in its parent's Children property, figures do not appear in the root's CurrentFigure property, objects do not appear in the root's CallbackObject property or in the figure's CurrentObject property, and axes do not appear in their parent's CurrentAxes property.

Overriding Handle Visibility

You can set the root ShowHiddenHandles property to on to make all handles visible regardless of their HandleVisibility settings. This does not affect the values of the HandleVisibility properties. See also findall.

Handle Validity

Hidden handles are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`
: `{on} | off`

Selectable by mouse click. Determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the line objects that compose the contour plot. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`HitTestArea`
: `on | {off}`

Select the object by clicking lines or area of extent. Select plot objects by:

- Clicking contour lines(default).
- Clicking anywhere in the extent of the plot.

When `HitTestArea` is `off`, you must click the contour lines (excluding the baseline) to select the object. When `HitTestArea` is `on`, you can select this object by clicking anywhere within the extent of the plot (that is, anywhere within a rectangle that encloses all the contour lines).

`Interruptible`
: `off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the `Interruptible` property affects only the callbacks for the`ButtonDownFcn` property. A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both the callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see "Control Callback Execution and Interruption".

`LabelSpacing`
distance in points (default = `144`)

Spacing between labels on each contour line. When you display contour line labels using either the `ShowText` property or the `clabel` command, the labels are spaced `144` points (2 inches) apart on each line. You can specify the spacing by setting the `LabelSpacing` property to a value in points. If the length of an individual contour line is less than the specified value, MATLAB displays only one contour label on that line.

`LevelList`
vector of `ZData`-values

Values at which contour lines are drawn. When the `LevelListMode` property is `auto`, the `contour` function automatically chooses contour values that span the range of values in `ZData` (the input argument `Z`). You can set this property to the values at which you want contour lines drawn.

To specify the contour interval (space between contour lines) use the `LevelStep` property.

`LevelListMode`
`{auto} | manual`

User-specified or autogenerated `LevelList` values. By default, the `contour` function automatically generates the values at which contours are drawn. If you set this property to `manual`, contour does not change the values in `LevelList` as you change the values of `ZData`.

`LevelStep`
scalar

Spacing of contour lines. The `contour` function draws contour lines at regular intervals determined by the value of `LevelStep`. When the `LevelStepMode` property is `auto`, `contour` determines the contour interval automatically based on the `ZData`.

`LevelStepMode`
`{auto} | manual`

User-specified or autogenerated `LevelStep` values. By default, the `contour` function automatically determines a value for the `LevelStep` property. If you set this property to `manual`, `contour` does not change the value of `LevelStep` as you change the values of `ZData`.

`LineColor`
`{auto} | ColorSpec | none`

Color of the contour lines. This property determines how MATLAB colors the contour lines.

- `auto` — Each contour line is a single color determined by its contour value, the figure colormap, and the color axis (`caxis`).
- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for edges. The default value is `[0 0 0]` (`black`). See `ColorSpec` for more information on specifying color.
- `none` — No contour lines are drawn.

`LineStyle`

`{-} | -- | :  | -.  | none`

Line style of contourgroup object.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line
'none'	No line

`LineWidth`

size in points

Width of linear objects and edges of filled areas. Specify in points. 1 point = $^1/_{72}$ inch. The default is `0.5` points.

`Parent`

handle of parent axes, hggroup, or hgtransform

Parent of object. Handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`

`on | {off}`

Object selection state. When you set this property to `on`, MATLAB displays selection handles at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You

can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
`{on} | off`

Object highlighted when selected.

- `on` — MATLAB indicates the selected state by drawing four edge handles and four corner handles.
- `off` — MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`ShowText`
`on | {off}`

Display labels on contour lines. When you set this property to `on`, MATLAB displays text labels on each contour line indicating the contour value. See also `LevelList`, `clabel`, and the example “Contour Graph of a Function” on page 1-1034.

`Tag`
string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

For example, create an areaseries object and set the `Tag` property.

```
t = area(Y,'Tag','area1')
```

When you want to access objects of a given type, use `findobj` to find the object’s handle. The following statement changes the `FaceColor` property of the object whose `Tag` is `area1`.

```
set(findobj('Tag','area1'),'FaceColor','red')
```

`TextList`
vector

Contour values to label. This property contains the contour values where text labels are placed. By default, these values are the same as those contained in the `LevelList` property, which define where the contour lines are drawn. Note that there must be an equivalent contour line to display a text label.

For example, the following statements create and label a contour plot:

```
[c,h]=contour(peaks);
clabel(c,h)
```

You can get the `LevelList` property to see the contour line values:

```
get(h,'LevelList')
```

Suppose you want to view the contour value `4.375` instead of the value of `4` that the contour function used. To do this, you need to set both the `LevelList` and `TextList` properties:

```
set(h,'LevelList',[-6 -4 -2 0 2 4.375 6 8],...
 'TextList',[-6 -4 -2 0 2 4.375 6 8])
```

See the example "Contour Graph of a Function" on page 1-1034 for additional information.

`TextListMode`
`{auto} | manual`

User-specified or auto `TextList` values. When this property is `auto`, MATLAB sets the `TextList` property equal to the values of the `LevelList` property (i.e., a text label for each contour line). When this property is `manual`, MATLAB does not set the values of the `TextList` property. Note that specifying values for the

`TextList` property causes the `TextListMode` property to be set to `manual`.

`TextStep`
scalar

Determines which contour line have numeric labels. The `contour` function labels contour lines at regular intervals which are determined by the value of the `TextStep` property. When the `TextStepMode` property is `auto`, `contour` labels every contour line when the `ShowText` property is `on`. See "Contour Graph of a Function" on page 1-1034 for an example that uses the `TextStep` property.

`TextStepMode`
`{auto} | manual`

User-specified or autogenerated `TextStep` values. By default, the `contour` function automatically determines a value for the `TextStep` property. If you set this property to `manual`, `contour` does not change the value of `TextStep` as you change the values of `ZData`.

`Type`
string (read-only)

Type of graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For contourgroup objects, `Type` is '`hggroup`'. This statement finds all the hggroup objects in the current axes.

```
t = findobj(gca,'Type','hggroup');
```

`UIContextMenu`
handle of uicontextmenu object

Associate context menu with object. Handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu`

function to create the context menu. MATLAB displays the context menu whenever you right-click over the object. The default value is an empty array.

`UserData`
array

User-specified data. Data you want to associate with this object (including cell arrays and structures). The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`
`{on} | off`

Visibility of object and its children.

- `on` — Object and all children of the object are visible unless the child object's `Visible` property is `off`.
- `off` — Object not displayed. However, the object still exists and you can set and query its properties.

`XData`
vector | matrix

x-axis values for graph. The *x*-axis values for graphs are specified by the `X` input argument. If `XData` is a vector, `length(XData)` must equal `length(YData)` and must be monotonic. If `XData` is a matrix, `size(XData)` must equal `size(YData)` and each column must be monotonic.

You can use `XData` to define meaningful coordinates for an underlying surface whose topography is being mapped. See "Changing the Offset of a Contour" for more information.

`XDataMode`
`{auto} | manual`

Use automatic or user-specified x-axis values. If you specify XData (by setting the XData property or specifying the X input argument), MATLAB sets this property to manual and uses the specified values to label the *x*-axis.

If you set XDataMode to auto after having specified XData, MATLAB resets the *x*-axis ticks to the column indices of the ZData, overwriting any previous values for XData.

XDataSource

MATLAB variable, as a string

Link XData to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the XData. The default value is an empty array.

```
set(h,'XDataSource','xdatavariablename')
```

MATLAB requires a call to refreshdata when you set this property. Changing workspace variables used as an object's XDataSource does not change the object's XData values, but you can use refreshdata to force an update of the object's data. refreshdata also lets you specify that the data source variable be evaluated in the workspace of a function from which you call refreshdata.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

YData

scalar | vector | matrix

Y-axis limits. This property determines the *y*-axis limits used in the contour plot. If you do not specify a Y argument, the contour

function calculates *y*-axis limits based on the size of the input argument Z.

YData can be either a matrix equal in size to ZData or a vector equal in length to the number of columns in ZData.

Use YData to define meaningful coordinates for the underlying surface whose topography is being mapped. See "Changing the Offset of a Contour" for more information.

YDataMode
{auto} | manual

Use automatic or user-specified y-axis values. In auto mode (the default) the contour function automatically determines the *y*-axis limits. If you set this property to manual, specify a value for YData, or specify a Y argument, then contour sets this property to manual and does not change the axis limits.

If you set YDataMode to auto after having specified YData, MATLAB resets the *y*-axis ticks to the row indices of the ZData, overwriting any previous values for YData.

YDataSource
MATLAB variable, as a string

Link YData to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the YData. The default value is an empty array.

```
set(h,'YDataSource','Ydatavariablename')
```

MATLAB requires a call to refreshdata when you set this property. Changing workspace variables used as an object's YDataSource does not change the object's YData values, but you can use refreshdata to force an update of the object's data. refreshdata also lets you specify that the data source variable

be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`ZData`
matrix

Contour data. This property contains the data from which the contour lines are generated (specified as the input argument `Z`). `ZData` must be at least a `2`-by-`2` matrix. The number of contour levels and the values of the contour levels are chosen automatically based on the minimum and maximum values of `ZData`. The limits of the *x*- and *y*-axis are `[1:n]` and `[1:m]`, where `[m,n] = size(ZData)`.

`ZDataSource`
MATLAB variable, as a string

Link `ZData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `ZData`. The default value is an empty array.

```
set(h,'ZDataSource','zdatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `ZDataSource` does not change the object's `ZData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

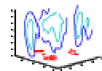
Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

How To

- "Contour Plots"
- "Core Graphics Objects"
- "Setting Default Property Values"
- "The Property Editor"

Purpose

Draw contours in volume slice planes



Syntax

```
contourslice(X,Y,Z,V,Sx,Sy,Sz)
contourslice(X,Y,Z,V,Xi,Yi,Zi)
contourslice(V,Sx,Sy,Sz)
contourslice(V,Xi,Yi,Zi)
contourslice(...,n)
contourslice(...,cvals)
contourslice(...,[cv cv])
contourslice(...,'method')
contourslice(axes_handle,...)
h = contourslice(...)
```

Description

`contourslice(X,Y,Z,V,Sx,Sy,Sz)` draws contours in the *x*-, *y*-, and *z*-axis aligned planes at the points in the vectors `Sx`, `Sy`, `Sz`. The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V` and must be monotonic and represent a Cartesian, axis-aligned grid (such as the data produced by `meshgrid`). The color at each contour is determined by the volume `V`, which must be an m-by-n-by-p volume array.

`contourslice(X,Y,Z,V,Xi,Yi,Zi)` draws contours through the volume `V` along the surface defined by the 2-D arrays `Xi,Yi,Zi`. The surface should lie within the bounds of the volume.

`contourslice(V,Sx,Sy,Sz)` and `contourslice(V,Xi,Yi,Zi)` (omitting the `X`, `Y`, and `Z` arguments) assume `[X,Y,Z] = meshgrid(1:n,1:m,1:p)`, where `[m,n,p]= size(v)`.

`contourslice(...,n)` draws `n` contour lines per plane, overriding the automatic value.

`contourslice(...,cvals)` draws `length(cval)` contour lines per plane at the values specified in vector `cvals`.

`contourslice(...,[cv cv])` computes a single contour per plane at the level `cv`.

`contourslice(...,'method')` specifies the interpolation method to use. *method* can be `linear`, `cubic`, or `nearest`. `nearest` is the default except when the contours are being drawn along the surface defined by `Xi`, `Yi`, `Zi`, in which case `linear` is the default. (See `interp3` for a discussion of these interpolation methods.)

`contourslice(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

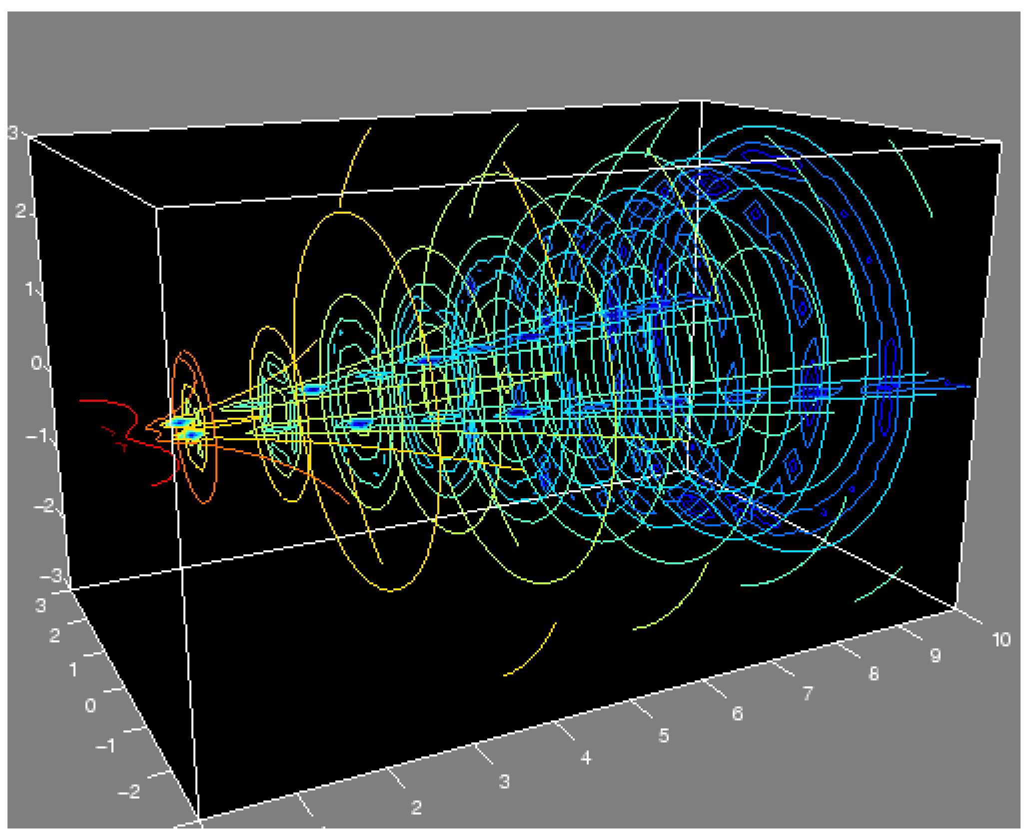
`h = contourslice(...)` returns a vector of handles to `patch` objects that are used to implement the contour lines.

Examples

This example uses the `flow` data set to illustrate the use of contoured slice planes. (Type `doc flow` for more information on this data set.) Notice that this example

- Specifies a vector of `length = 9` for `Sx`, an empty vector for the `Sy`, and a scalar value (0) for `Sz`. This creates nine contour plots along the x direction in the y-z plane, and one in the x-y plane at z = 0.
- Uses `linspace` to define a 10-element vector of linearly spaced values from -8 to 2. This vector specifies that 10 contour lines be drawn, one at each element of the vector.
- Defines the view and projection type (`camva`, `camproj`, `campos`).
- Sets figure (`gcf`) and axes (`gca`) characteristics.

```
[x y z v] = flow;
h = contourslice(x,y,z,v,[1:9],[],[0],linspace(-8,2,10));
axis([0,10,-3,3,-3,3]); daspect([1,1,1])
camva(24); camproj perspective;
campos([-3,-15,5])
set(gcf,'Color',[.5,.5,.5],'Renderer','zbuffer')
set(gca,'Color','black','XColor','white', ...
 'YColor','white','ZColor','white')
box on
```



This example draws contour slices along a spherical surface within the volume.

```
[x,y,z] = meshgrid(-2:.2:2,-2:.25:2,-2:.16:2);
v = x.*exp(-x.^2-y.^2-z.^2); % Create volume data
[xi,yi,zi] = sphere; % Plane to contour
contourslice(x,y,z,v,xi,yi,zi)
view(3)
```

See Also

isosurface | slice | smooth3 | subvolume | reducevolume

Purpose

Grayscale colormap for contrast enhancement

Syntax

```
cmap = contrast(X)
cmap = contrast(X,m)
```

Description

The `contrast` function enhances the contrast of an `image`. It creates a new gray colormap, `cmap`, that has an approximately equal intensity distribution. All three elements in each row are identical.

`cmap = contrast(X)` returns a gray colormap that is the same length as the current colormap. If there are `NaN` or `Inf` elements in `X` the length of the colormap increases.

`cmap = contrast(X,m)` returns an `m`-by-3 gray colormap.

Examples

Add contrast to the clown image defined by `X`.

```
load clown;
cmap = contrast(X);
image(X);
colormap(cmap);
```

See Also

`brighten` | `colormap` | `image`

Purpose

Convolution and polynomial multiplication

Syntax

```
w = conv(u,v)
w = conv(...,'shape')
```

Description

`w = conv(u,v)` convolves vectors `u` and `v`. Algebraically, convolution is the same operation as multiplying the polynomials whose coefficients are the elements of `u` and `v`.

`w = conv(...,'shape')` returns a subsection of the convolution, as specified by the `shape` parameter:

`full`	Returns the full convolution (default).
`same`	Returns the central part of the convolution of the same size as `u`.
`valid`	Returns only those parts of the convolution that are computed without the zero-padded edges. Using this option, `length(w)` is `max(length(u)-max(0,length(v)-1),0)`.

Definitions

Let `m = length(u)` and `n = length(v)` . Then `w` is the vector of length `m+n-1` whose `k`th element is

$$w(k) = \sum_j u(j)v(k-j+1)$$

The sum is over all the values of `j` which lead to legal subscripts for `u(j)` and `v(k+1-j)`, specifically `j = max(1,k+1-n): min(k,m)`. When `m = n`, this gives

```
w(1) = u(1)*v(1)
w(2) = u(1)*v(2)+u(2)*v(1)
w(3) = u(1)*v(3)+u(2)*v(2)+u(3)*v(1)
...
w(n) = u(1)*v(n)+u(2)*v(n-1)+ ... +u(n)*v(1)
...
```

```
w(2*n-1) = u(n)*v(n)
```

See Also

conv2 | convn | deconv | filter | convmtx | xcorr

Purpose

2-D convolution

Syntax

```
C = conv2(A,B)
C = conv2(h1,h2,A)
C = conv2(...,shape)
```

Description

`C = conv2(A,B)` computes the two-dimensional convolution of matrices `A` and `B`. If one of these matrices describes a two-dimensional finite impulse response (FIR) filter, the other matrix is filtered in two dimensions. The size of `C` is determined as follows: if `[ma,na] = size(A)`, `[mb,nb] = size(B)`, and `[mc,nc] = size(C)`, then `mc = max([ma+mb-1,ma,mb])` and `nc = max([na+nb-1,na,nb])`.

`C = conv2(h1,h2,A)` first convolves `A` with the vector `h1` along the rows and then with the vector `h2` along the columns. The size of `C` is determined as follows: if `n1 = length(h1)` and `n2 = length(h2)`, then `mc = max([ma+n1-1,ma,n1])` and `nc = max([na+n2-1,na,n2])`.

`C = conv2(...,shape)` returns a subsection of the two-dimensional convolution, as specified by the `shape` parameter:

`'full'`	Returns the full two-dimensional convolution (default).
`'same'`	Returns the central part of the convolution of the same size as `A`.
`'valid'`	Returns only those parts of the convolution that are computed without the zero-padded edges. Using this option, `size(C) = max([ma-max(0,mb-1),na-max(0,nb-1)],0)`.

Note All numeric inputs to `conv2` must be of type `double` or `single`.

Algorithms

`conv2` uses a straightforward formal implementation of the two-dimensional convolution equation in spatial form. If a and b are

functions of two discrete variables, n_1 and n_2, then the formula for the two-dimensional convolution of a and b is

$$c(n_1,n_2) = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} a(k_1,k_2) b(n_1-k_1,n_2-k_2)$$

In practice however, `conv2` computes the convolution for finite intervals.

Note that matrix indices in MATLAB software always start at 1 rather than 0. Therefore, matrix elements `A(1,1)`, `B(1,1)`, and `C(1,1)` correspond to mathematical quantities $a(0,0)$, $b(0,0)$, and $c(0,0)$.

Examples

Example 1

For the `'same'` case, `conv2` returns the central part of the convolution. If there are an odd number of rows or columns, the "center" leaves one more at the beginning than the end.

This example first computes the convolution of A using the default (`'full'`) shape, then computes the convolution using the `'same'` shape. Note that the array returned using `'same'` corresponds to the red highlighted elements of the array returned using the default shape.

```
A = rand(3);
B = rand(4);
C = conv2(A,B)   % C is 6-by-6

C =
   0.1838   0.2374   0.9727   1.2644   0.7890   0.3750
   0.6929   1.2019   1.5499   2.1733   1.3325   0.3096
   0.5627   1.5150   2.3576   3.1553   2.5373   1.0602
   0.9986   2.3811   3.4302   3.5128   2.4489   0.8462
   0.3089   1.1419   1.8229   2.1561   1.6364   0.6841
   0.3287   0.9347   1.6464   1.7928   1.2422   0.5423

Cs = conv2(A,B,'same')   % Cs is the same size as A: 3-by-3
Cs =
   2.3576   3.1553   2.5373
```

```
    3.4302   3.5128   2.4489
    1.8229   2.1561   1.6364
```

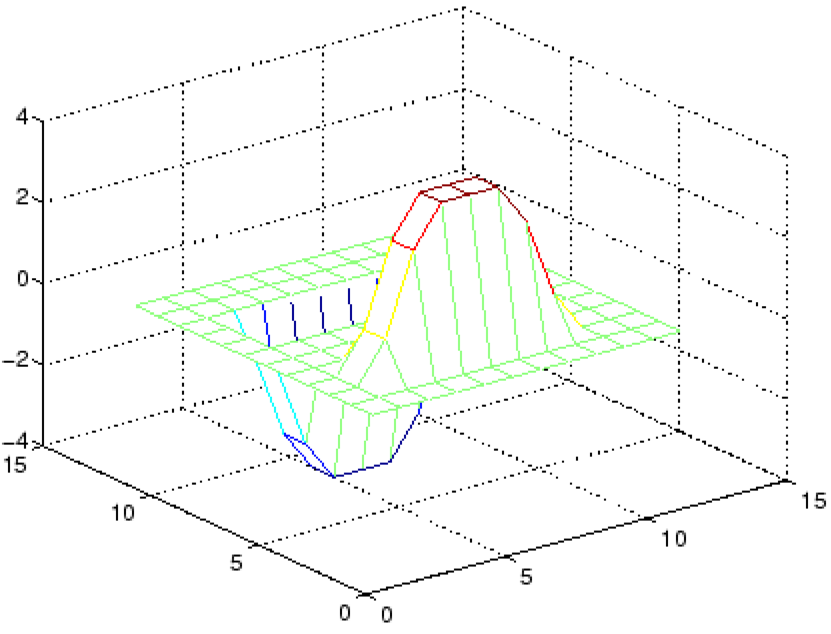
Example 2

In image processing, the Sobel edge finding operation is a two-dimensional convolution of an input array with the special matrix

```
s = [1 2 1; 0 0 0; -1 -2 -1];
```

These commands extract the horizontal edges from a raised pedestal.

```
A = zeros(10);
A(3:7,3:7) = ones(5);
H = conv2(A,s);
mesh(H)
```

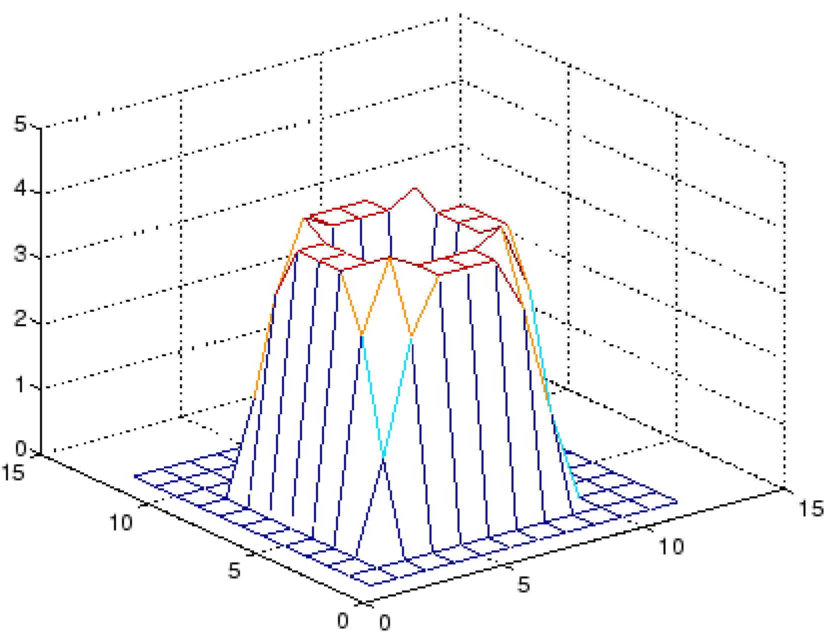


Transposing the filter s extracts the vertical edges of A.

```
V = conv2(A,s');
figure, mesh(V)
```

This figure combines both horizontal and vertical edges.

```
figure
mesh(sqrt(H.^2 + V.^2))
```



See Also

`conv` | `convn` | `filter2` | `xcorr2`

convhull

Purpose

Convex hull

Note Qhull-specific options are no longer supported. Remove the `OPTIONS` argument from all instances in your code that pass it to `convhull`.

Syntax

```
K = convhull(X,Y)
K = convhull(X,Y,Z)
K = convhull(X)
K = convhull(...,'simplify', logicalvar)
[K,V] = convhull(...)
```

Definitions

`convhull` returns the convex hull of a set of points in 2-D or 3-D space.

Description

`K = convhull(X,Y)` returns the 2-D convex hull of the points (`X`,`Y`), where `X` and `Y` are column vectors. The convex hull `K` is expressed in terms of a vector of point indices arranged in a counterclockwise cycle around the hull.

`K = convhull(X,Y,Z)` returns the 3-D convex hull of the points (`X`,`Y`,`Z`), where `X`, `Y`, and `Z` are column vectors. `K` is a triangulation representing the boundary of the convex hull. `K` is of size `mtri`-by-3, where `mtri` is the number of triangular facets. That is, each row of `K` is a triangle defined in terms of the point indices.

`K = convhull(X)` returns the 2-D or 3-D convex hull of the points `X`. This variant supports the definition of points in matrix format. `X` is of size `mpts`-by-`ndim`, where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside, $2 \leqq$ `ndim` $\leqq 3$. The output facets are equivalent to those generated by the 2-input or 3-input calling syntax.

`K = convhull(...,'simplify', logicalvar)` provides the option of removing vertices that do not contribute to the area/volume of the convex hull, the default is false. Setting `'simplify'` to true returns the topology in a more concise form.

`[K,V] = convhull(...)` returns the convex hull `K` and the corresponding area/volume `V` bounded by `K`.

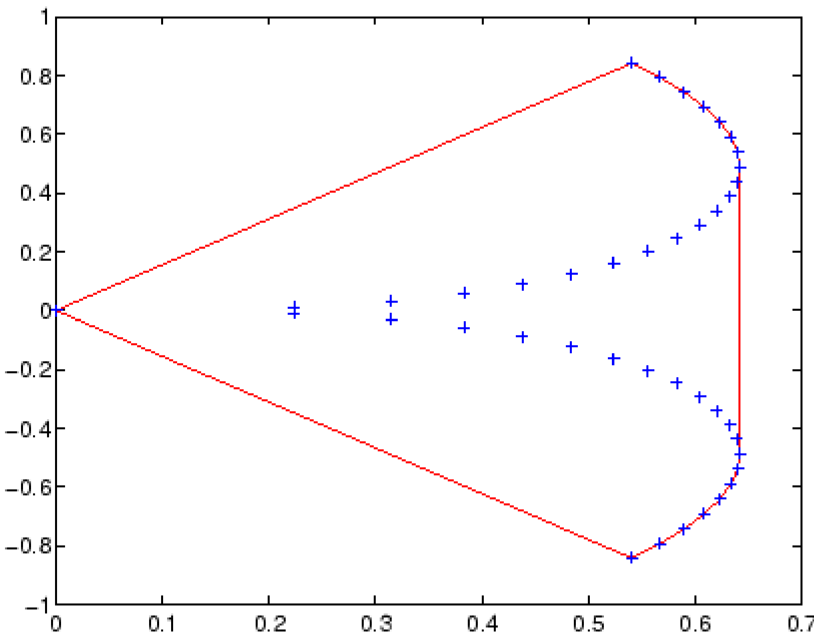
Visualization

Use `plot` to plot the output of `convhull` in 2-D. Use `trisurf` or `trimesh` to plot the output of convhull in 3-D.

Examples

Example 1

```
xx = -1:.05:1; yy = abs(sqrt(xx));
[x,y] = pol2cart(xx,yy);
k = convhull(x,y);
plot(x(k),y(k),'r-',x,y,'b+')
```



See Also

`convexHull` | `voronoiDiagram` | `convhulln` | `delaunay` | `polyarea` | `voronoi`

convhulln

Purpose

N-D convex hull

Syntax

```
K = convhulln(X)
K = convhulln(X, options)
[K, v] = convhulln(...)
```

Description

`K = convhulln(X)` returns the indices `K` of the points in `X` that comprise the facets of the convex hull of `X`. `X` is an `m`-by-`n` array representing `m` points in N-dimensional space. If the convex hull has `p` facets then `K` is `p`-by-`n`.

`convhulln` uses Qhull.

`K = convhulln(X, options)` specifies a cell array of strings `options` to be used as options in Qhull. The default options are:

- `{'Qt'}` for 2-, 3-. and 4-dimensional input
- `{'Qt','Qx'}` for 5-dimensional input and higher.

If `options` is `[]`, the default options are used. If `options` is `{''}`, no options are used, not even the default. For more information on Qhull and its options, see `http://www.qhull.org/`.

`[K, v] = convhulln(...)` also returns the volume `v` of the convex hull.

Visualization

Plotting the output of `convhulln` depends on the value of `n`:

- For `n = 2`, use `plot` as you would for `convhull`.
- For `n = 3`, you can use `trisurf` to plot the output. The calling sequence is

  ```
  K = convhulln(X);
  trisurf(K,X(:,1),X(:,2),X(:,3))
  ```

- You cannot plot `convhulln` output for `n > 3`.

Examples

The following example illustrates the `options` input for `convhulln`. The following commands

```
X = [0 0; 0 1e-10; 0 0; 1 1];
K = convhulln(X)
```

return a warning.

```
Warning: qhull precision warning:
The initial hull is narrow
(cosine of min. angle is 0.9999999999999998).
A coplanar point may lead to a wide facet.
Options 'QbB' (scale to unit box) or 'Qbb'
(scale last coordinate) may remove this warning.
Use 'Pp' to skip this warning.
```

To suppress the warning, use the option `'Pp'`. The following command passes the option `'Pp'`, along with the default `'Qt'`, to `convhulln`.

```
K = convhulln(X,{'Qt','Pp'})

K =

     1     4
     1     2
     4     2
```

Algorithms

`convhulln` is based on Qhull [1]. For information about Qhull, see `http://www.qhull.org/`. For copyright information, see `http://www.qhull.org/COPYING.txt`.

References

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, “The Quickhull Algorithm for Convex Hulls,” ACM *Transactions on Mathematical Software*, Vol. 22, No. 4, Dec. 1996, p. 469-483.

See Also

`convexHull` | `convhull` | `delaunayn` | `dsearchn` | `tsearchn` | `voronoin`

convn

Purpose

N-D convolution

Syntax

```
C = convn(A,B)
C = convn(A,B,'shape')
```

Description

`C = convn(A,B)` computes the N-dimensional convolution of the arrays `A` and `B`. The size of the result is `size(A)+size(B)-1`.

`C = convn(A,B,'shape')` returns a subsection of the N-dimensional convolution, as specified by the `shape` parameter:

`'full'`	Returns the full N-dimensional convolution (default).
`'same'`	Returns the central part of the result that is the same size as `A`.
`'valid'`	Returns only those parts of the convolution that can be computed without assuming that the array A is zero-padded. The size of the result is `max(size(A)-size(B) + 1, 0)`

See Also

`conv` | `conv2`

Purpose

Superclass providing copy functionality for handle objects

Description

The `matlab.mixin.Copyable` class is an abstract handle class that provides a `copy` method for copying handle objects. The `copy` method makes a shallow copy of the object (that is, it shallow copies all non-dependent properties from the source to the destination object). In making a shallow copy, MATLAB does not call `copy` recursively on any handles contained in property values.

Subclass `matlab.mixin.Copyable` when you want to define handles classes that inherit a `copy` method. The `copy` method:

- Copies data without calling class constructors or property set functions and therefore produces no side effects.
- Enables subclasses to customize the copy behavior

Customizing Subclass Copy Behavior

The `copy` method provides the public, non-overrideable interface to copy behavior. `copy` takes an array of objects as input and returns an array of the same shape and dimensions.

`copyElement` is a protected method that the `copy` method uses to perform the copy operation on each object in the input array. `copyElement` is not `Sealed` so you can override it in your subclass to customize the behavior of the inherited `copy` method.

Implementing a Selective Deep Copy

This example overrides the `copyElement` method in a subclass of `matlab.mixin.Copyable` to implement a deep copy of a specific class of `handle` objects.

Consider the following classes:

- `ContainsHandles` — subclass of `matlab.mixin.Copyable` that contains `handle` objects in two properties
- `DeepCp` — subclass of `matlab.mixin.Copyable`
- `ShallowCp` — subclass of `handle`

Here are the simplified class definitions:

```
classdef ContainsHandles < matlab.mixin.Copyable
   properties
      Prop1
      Prop2
      DeepObj     % Contains a DeepCp object
      ShallowObj  % Contains a ShallowCp object
   end
   methods
      function obj = ContainsHandles(val1, val2, deepobj, shallowobj)
         if nargin > 0
            obj.Prop1 = val1;
            obj.Prop2 = val2;
            obj.DeepObj = deepobj;
            obj.ShallowObj = shallowobj;
          end
      end
   end
      methods(Access = protected)
       % Override copyElement method:
      function cpObj = copyElement(obj)
         % Make a shallow copy of all four properties
         cpObj = copyElement@matlab.mixin.Copyable(obj);
         % Make a deep copy of the DeepCp object
         cpObj.DeepObj = copy(obj.DeepObj);
      end
   end
end
```

The `DeepCp` class derives from `matlab.mixin.Copyable`:

```
classdef DeepCp < matlab.mixin.Copyable
   properties
      DpProp
   end
   methods
      function obj = DeepCp(val)
```

```
      ...
      end
   end
end
```

The handle class ShallowCp does not derive from matlab.mixin.Copyable and, therefore, has no copy method:

```
classdef ShallowCp < handle
   properties
      ShProp
   end
   methods
      function obj = ShallowCp(val)
       ...
      end
   end
end
```

Create a ContainsHandles object, which contains the two handle objects in its DpProp and ShProp properties:

```
>> sc = ShallowCp(7);
>> dc = DeepCp(7);
>> a = ContainsHandles(4,5,dc,sc);
>> a.DeepObj
ans =

  DeepCp with properties:

    DpProp: 7
>> a.ShallowObj.ShProp
ans =

  ShallowCp with properties:

    ShProp: 7
```

Make a copy of the `ContainsHandles` object:

```
>> b = copy(a);
```

The returned copy `b` contains a shallow copy of object `sc`, and a deep copy of object `dc`. That is, the `dc` object passed to `ContainsHandles` constructor is now a new, independent objects as a result of the copy operation. You can now change the `dc` object without affecting the copy. This is not the case for the shallow copied object, `sc`:

```
% Change the property values of the handle objects:
>> sc.ShProp = 5;
>> dc.DpProp = 5;
% Note that the deep copied object is not affected:
>> b.DeepObj

ans =

  DeepCp with properties:

    DpProp: 7
% The shallow copied object is still referencing the same data:
>> b.ShallowObj

ans =

  ShallowCp with properties:

    ShProp: 5
```

Overriding Copy Behavior in Hierarchies

The `copyElement` method in a superclass cannot access the private data in a subclass.

If you override `copyElement` in a subclass of `matlab.mixin.Copyable`, and then use this subclass as a superclass, you need to override `copyElement` in all subclasses that contain private properties. The

override of `copyElement` in subclasses should call the `copyElement` in the respective superclass, as in the previous example.

The following simplified code demonstrates this approach:

```
classdef SuperClass < matlab.mixin.Copyable
   properties(Access = private)
      super_prop
   end
   methods
        ...

      function cpObj = copyElement(obj)
            ...
         cpObj = copyElement@mixin.matlab.Copyable(obj);
            ...
      end
   end
end


classdef SubClass1 < SuperClass
   properties(Access=private)
      sub_prop1
   end
   methods
      function cpObj = copyElement(obj)
         % Copy super_prop
         cpObj = copyElement@SuperClass(obj);
         % Copy sub_prop1 in subclass
         % Assignment can introduce side effects
         cpObj.sub_prop1 = obj.sub_prop1;
      end
   end
end
```

The override of `copyElement` in `SubClass1` copies the private subclass property because the superclass cannot access private data in the subclass.

Note The assignment of `sub_prop1` in the override of `copyElement` in `SubClass1` calls the property set method, if one exists, possibly introducing side effects to the copy operation.

Copy Behaviors for Specific Inputs

Given a call to the `matlab.mixin.Copyable` `copy` method of the form:

```
B = copy(A);
```

Under the following conditions, produces the described results:

- `A` has dynamic properties — `copy` does not copy dynamic properties. You can implement dynamic-property copying in the subclass if needed.
- `A` has no non-Dependent properties — `copy` creates a new object with no property values without calling the class constructor to avoid introducing side effects.
- `A` contains deleted handles — `copy` creates deleted handles of the same class in the output array.
- `A` has attached listeners — `copy` does not copy listeners.
- `A` contains objects of enumeration classes — Enumeration classes cannot subclass `matlab.mixin.Copyable`.
- A `delete` method calls `copy` — `copy` creates a legitimate copy, obeying all the behaviors that apply in any other usage.

Note You cannot derive an enumeration class from `matlab.mixin.Copyable` because the number of instances you can create are limited to the ones defined inside the enumeration block. See "Working with Enumerations" for more information about enumeration classes.

Methods

copy	Copy array of handle objects

Definitions

Deep Copy

Copy each property value and assign it to the new (copied) property. Recursively copy property values that reference handle objects to copy all of the underlying data.

Shallow Copy

Copy each property value and assign it to the new (copied) property. If a property value is a handle, copy the handle but not the underlying data.

Attributes

`ConstructOnLoad`	`true`

To learn about attributes of classes, see Class Attributes in the MATLAB Object-Oriented Programming documentation.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

See Also

`handle`

How To

- Class Attributes

matlab.mixin.Copyable.copy

Purpose

Copy array of handle objects

Syntax

`B = copy(A)`

Description

`B = copy(A)` copies each element in the array of handles `A` to the new array of handles `B`.

The `copy` method performs a copy according to the following rules:

- The `copy` method does not copy `Dependent` properties
- MATLAB does not call the `copy` method recursively on any handles contained in property values
- MATLAB does not call the class constructor or property set methods during the copy operation.
- `B` has the same number of elements and same size as `A`.
- `B` is the same class as `A`.
- If `A` is empty, `B` is also empty.
- If `A` is heterogeneous, `B` is also heterogeneous.
- If `A` contains deleted handle objects, copy creates deleted handles of the same class in `B`.
- Dynamic properties and listeners associated with objects in `A` are not copied to objects in `B`.
- You can call `copy` inside your class `delete` method.

Input Arguments

A

Handle object array

Output Arguments

B

Handle object array containing copies of the objects in `A`.

Attributes

Sealed	true

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

matlab.mixin.Copyable | handle | copyElement

copyfile

Purpose

Copy file or folder

Syntax

```
copyfile('source','destination')
copyfile('source','destination','f')
[status] = copyfile( ___ )
[status, message] = copyfile( ___ )
[status,message,messageid] = copyfile( ___ )
```

Description

`copyfile('source','destination')` copies the file or folder named `source` to the file or folder `destination`. The values for `source` and `destination` are 1 x n strings. Use full path names or path names relative to the current folder. To copy multiple files or folders, use one or more wildcard characters (`*`) after the last file separator in `source`. You cannot use a wildcard character in `destination`.

`copyfile('source','destination','f')` copies `source` to `destination`, even when `destination` is not writable. The state of the read-write attribute for `destination` does not change. You can use `f` with any syntax for `copyfile`.

`[status] = copyfile( ___ )` reports the outcome as a logical scalar, `status`. The value is `1` for success and `0` for failure.

`[status, message] = copyfile( ___ )` returns any warning or error message as a string to `message`. When `copyfile` succeeds, `message` is an empty string.

`[status,message,messageid] = copyfile( ___ )` returns any warning or error identifier as a string to `messageId`. When `copyfile` succeeds, `messageId` is an empty string.

Tips

- The timestamp for `destination` is the same as the timestamp for `source`.
- When `source` is a folder, `destination` must be a folder.
  - When `source` is a folder and `destination` does not exist, `copyfile` creates `destination` and copies the contents of `source` into `destination`.

- When `source` is a folder and `destination` is an existing folder, `copyfile` copies the contents of `source` into `destination`.
- When `source` is multiple files and `destination` does not exist, `copyfile` creates `destination`.

Examples

Copy File to Another Folder

Copy `myFun.m` from the current folder to `d:/work/Projects/`.

```
copyfile('myFun.m','d:/work/Projects/')
```

Copy File to Its Current Folder

Copy `myFun.m` in the current folder, assigning it the name `myFun2.m`.

```
copyfile('myFun.m','myFun2.m')
```

Copy Files and Folders to a New Folder Using Wildcards

Copy files and subfolders whose names begin with `my`, from the `Projects` subfolder within the current folder to the folder `newProjects`, which is at the same level as the current folder:

```
copyfile('Projects/my*','../newProjects/')
```

Copy Files to a New, Nonexistent Folder

Copy the contents of the `Projects` subfolder within the current folder to the `I:/work/newProjects` folder, where `newProjects` does not exist.

```
copyfile('Projects','I:/work/newProjects')
```

Copy File that Overwrites a Read-Only File

Copy the contents of `myFun.m` from the current folder to `d:/work/restricted/myFun2.m`, where `myFun2.m` is read-only.

```
[status,message,messageId]=copyfile('myFun.m', ...
'd:/work/restricted/myFun2.m','f')
```

```
status =
     1

message =
     ''

messageId =
     ''
```

The `status` of 1 and empty `message` and `messageId` strings confirm the copy was successful.

See Also

`cd` | `delete` | `dir` | `fileattrib` | `filebrowser` | `fileparts` | `mkdir` | `movefile` | `rmdir`

How To

- "Path Names in MATLAB"
- "Creating, Opening, Changing, and Deleting Files and Folders"

Purpose

Copy graphics objects and their descendants

Syntax

```
new_handle = copyobj(h,p)
```

Description

`copyobj` creates copies of graphics objects. The copies are identical to the original objects except the copies have different values for their `Parent` property and a new handle. The new parent must be appropriate for the copied object (e.g., you can copy a line object only to another axes object).

`new_handle = copyobj(h,p)` copies one or more graphics objects identified by `h` and returns the handle of the new object or a vector of handles to new objects. The new graphics objects are children of the graphics objects specified by `p`.

Tips

`h` and `p` can be scalars or vectors. When both are vectors, they must be the same length, and the output argument, `new_handle,` is a vector of the same length. In this case, `new_handle(i)` is a copy of `h(i)` with its `Parent` property set to `p(i)`.

When `h` is a scalar and `p` is a vector, `h` is copied once to each of the parents in `p`. Each `new_handle(i)` is a copy of `h` with its `Parent` property set to `p(i)`, and `length(new_handle)` equals `length(p)`.

When `h` is a vector and `p` is a scalar, each `new_handle(i)` is a copy of `h(i)` with its `Parent` property set to `p`. The length of `new_handle` equals `length(h)`.

When programming a GUI, do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

Examples

Copy a surface to a new axes within a different figure.

```
h = surf(peaks);
colormap hot
figure    % Create a new figure
```

```
axes    % Create an axes object in the figure
new_handle = copyobj(h,gca);
colormap hot
view(3)
grid on
```

Note that while the surface is copied, the `colormap` (`figure` property), `view`, and `grid` (`axes` properties) are not copied.

See Also

`findobj` | `gcf` | `gca` | `gco` | `get` | `set`

Purpose

Correlation coefficients

Syntax

```
R = corrcoef(X)
R = corrcoef(x,y)
[R,P]=corrcoef(...)
[R,P,RLO,RUP]=corrcoef(...)
[...]=corrcoef(...,'param1',val1,'param2',val2,...)
```

Description

`R = corrcoef(X)` returns a matrix `R` of correlation coefficients calculated from an input matrix `X` whose rows are observations and whose columns are variables. The matrix `R = corrcoef(X)` is related to the covariance matrix `C = cov(X)` by

$$R(i,j) = \frac{C(i,j)}{\sqrt{C(i,i)C(j,j)}}.$$

`corrcoef(X)` is the zeroth lag of the normalized covariance function, that is, the zeroth lag of `xcov(x,'coeff')` packed into a square array.

`R = corrcoef(x,y)` where `x` and `y` are column vectors is the same as `corrcoef([x y])`. If `x` and `y` are not column vectors, `corrcoef` converts them to column vectors. For example, in this case `R=corrcoef(x,y)` is equivalent to `R=corrcoef([x(:) y(:)])`.

`[R,P]=corrcoef(...)` also returns `P`, a matrix of p-values for testing the hypothesis of no correlation. Each p-value is the probability of getting a correlation as large as the observed value by random chance, when the true correlation is zero. If `P(i,j)` is small, say less than `0.05`, then the correlation `R(i,j)` is significant.

`[R,P,RLO,RUP]=corrcoef(...)` also returns matrices `RLO` and `RUP`, of the same size as `R`, containing lower and upper bounds for a 95% confidence interval for each coefficient.

`[...]=corrcoef(...,'param1',val1,'param2',val2,...)` specifies additional parameters and their values. Valid parameters are the following.

'alpha'	A number between 0 and 1 to specify a confidence level of 100*(1 – alpha)%. Default is 0.05 for 95% confidence intervals.
'rows'	Either 'all' (default) to use all rows, 'complete' to use rows with no NaN values, or 'pairwise' to compute R(i,j) using rows with no NaN values in either column i or j.

The p-value is computed by transforming the correlation to create a t statistic having n-2 degrees of freedom, where n is the number of rows of X. The confidence bounds are based on an asymptotic normal distribution of 0.5*log((1+R)/(1-R)), with an approximate variance equal to 1/(n-3). These bounds are accurate for large samples when X has a multivariate normal distribution. The 'pairwise' option can produce an R matrix that is not positive definite.

Examples

Generate random data having correlation between column 4 and the other columns.

```
x = randn(30,4);     % Uncorrelated data
x(:,4) = sum(x,2);   % Introduce correlation.
[r,p] = corrcoef(x)  % Compute sample correlation and p-values.
[i,j] = find(p<0.05);  % Find significant correlations.
[i,j]                % Display their (row,col) indices.

r =
    1.0000   -0.3566    0.1929    0.3457
   -0.3566    1.0000   -0.1429    0.4461
    0.1929   -0.1429    1.0000    0.5183
    0.3457    0.4461    0.5183    1.0000

p =
    1.0000    0.0531    0.3072    0.0613
    0.0531    1.0000    0.4511    0.0135
    0.3072    0.4511    1.0000    0.0033
    0.0613    0.0135    0.0033    1.0000
```

```
ans =
     4     2
     4     3
     2     4
     3     4
```

See Also

cov | mean | median | std | var | xcorr | xcov

Purpose

Cosine of argument in radians

Syntax

```
Y = cos(X)
```

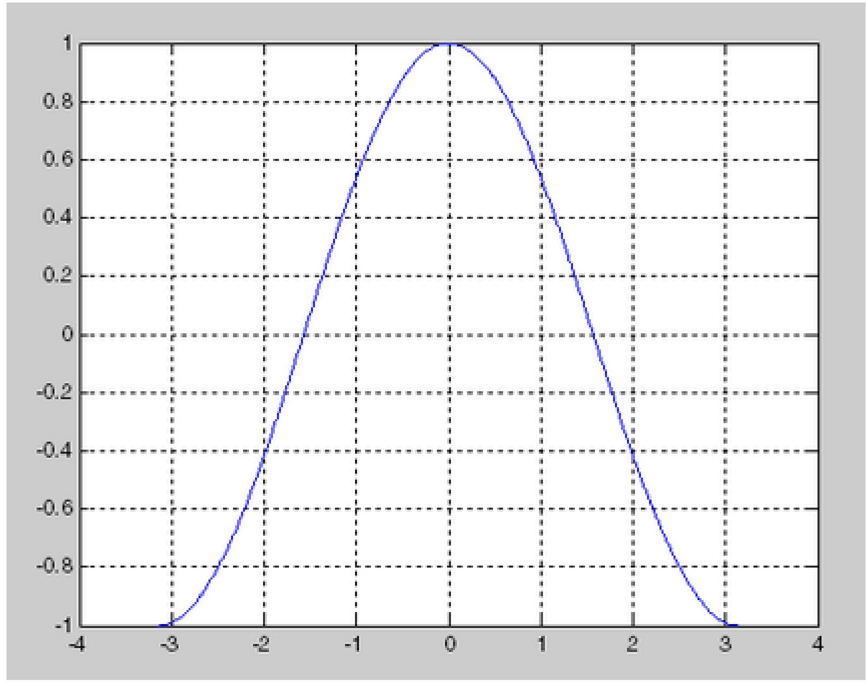
Description

`Y = cos(X)` returns the cosine for each element of `X`. The `cos` function operates element-wise on arrays. All angles are in radians.

Examples

Graph the cosine function over the domain $-\pi \le x \le \pi$:

```
x = -pi:0.01:pi;
plot(x,cos(x)), grid on
```



See Also

`cosd` | `acos` | `acosd` | `cosh`

Purpose

Cosine of argument in degrees

Syntax

```
Y = cosd(X)
```

Description

`Y = cosd(X)` returns the cosine of the elements of `X`, which are expressed in degrees.

Input Arguments

X - Angle in degrees

scalar value | vector | matrix | N-D array

Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `cosd` operation is element-wise when `X` is nonscalar.

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

Output Arguments

Y - Cosine of angle

scalar value | vector | matrix | N-D array

Cosine of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as `X`.

Examples

Cosine of 90 degrees compared to cosine of /2 radians

```
cosd(90)

ans =

     0

cos(pi/2)

ans =

   6.1232e-17
```

Cosine of complex angles specified in degrees

Create an array of three complex angles and compute the cosine.

```
z = [180+i 45+2i 10+3i];
y = cosd(z)

y =

  -1.0002          0.7075 - 0.0247i   0.9862 - 0.0091i
```

See Also

`cos` | `acos` | `acosd`

Purpose

Hyperbolic cosine

Syntax

```
Y = cosh(X)
```

Description

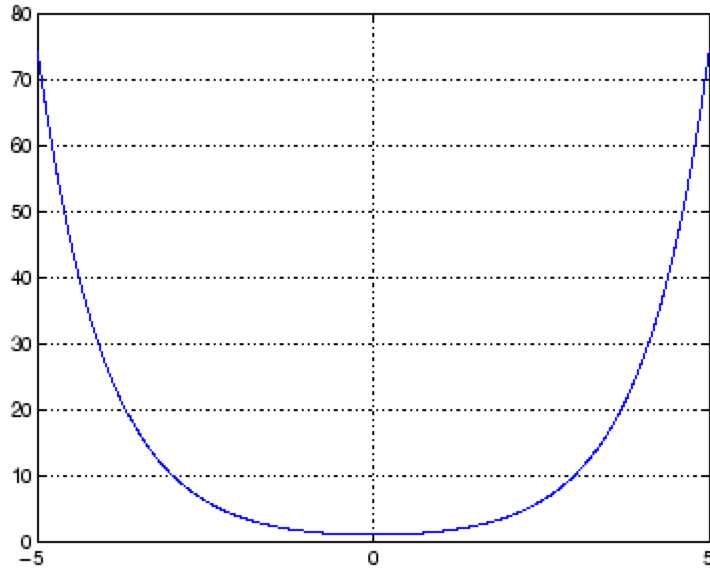
The `cosh` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

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

Examples

Graph the hyperbolic cosine function over the domain $-5 \le x \le 5$.

```
x = -5:0.01:5;
plot(x,cosh(x)), grid on
```



See Also

`acosh` | `cos` | `sinh` | `tanh`

Purpose

Cotangent of argument in radians

Syntax

```
Y = cot(X)
```

Description

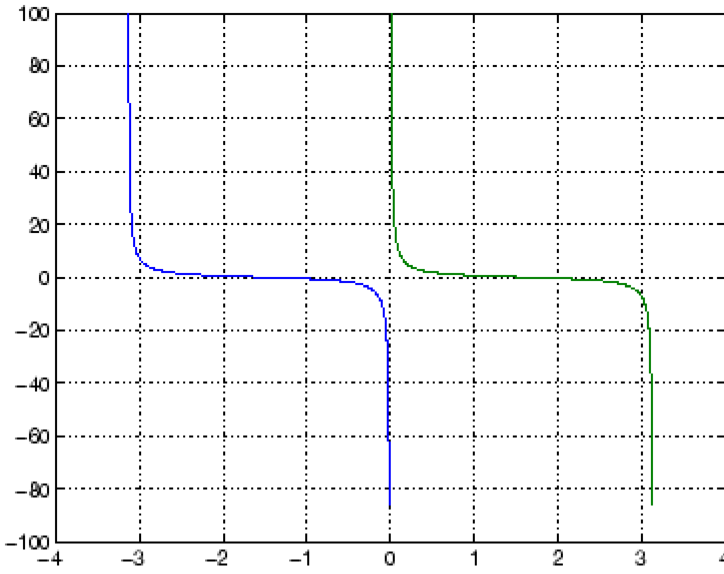
The `cot` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = cot(X)` returns the cotangent for each element of `X`.

Examples

Graph the cotangent the domains $-\pi < x < 0$ and $0 < x < \pi$.

```
x1 = -pi+0.01:0.01:-0.01;
x2 = 0.01:0.01:pi-0.01;
plot(x1,cot(x1),x2,cot(x2)), grid on
```



See Also

`cotd` | `coth` | `acot` | `acotd` | `acoth`

Purpose

Cotangent of argument in degrees

Syntax

```
Y = cotd(X)
```

Description

`Y = cotd(X)` returns the cotangent of the elements of `X`, which are expressed in degrees.

Input Arguments

X - Angle in degrees

scalar value | vector | matrix | N-D array

Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `cotd` operation is element-wise when `X` is nonscalar.

Data Types

`single` | `double`

Complex Number Support: Yes

Output Arguments

Y - Cotangent of angle

scalar value | vector | matrix | N-D array

Cotangent of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as `X`.

Examples

Cotangent of angles approaching 90 and 180 degrees

Create a vector of input angles consisting of 90° and the next smaller and larger double precision numbers. Then compute the cotangent.

```
x1 = [90-eps(90) 90 90+eps(90)];
y1 = cotd(x1)
```

```
y1 =

   1.0e-15 *

    0.2480         0   -0.2480
```

`cotd` returns zero when the input angle is exactly 90°. Evaluation at the next smaller double-precision angle returns a slightly positive result. Likewise, the cotangent is slightly negative when the input angle is the next double-precision number larger than 90.

The behavior is similar for input angles near 180°.

```
x2 = [180-eps(180) 180 180+eps(180)];
y2 = cotd(x2)

y2 =

   1.0e+15 *

   -2.0159       Inf    2.0159
```

Cotangent of complex angle, specified in degrees

```
x = 35+5i;
y = cotd(x)

y =

   1.3958 - 0.2606i
```

See Also

`acotd` | `cot` | `acot`

Purpose

Hyperbolic cotangent

Syntax

```
Y = coth(X)
```

Description

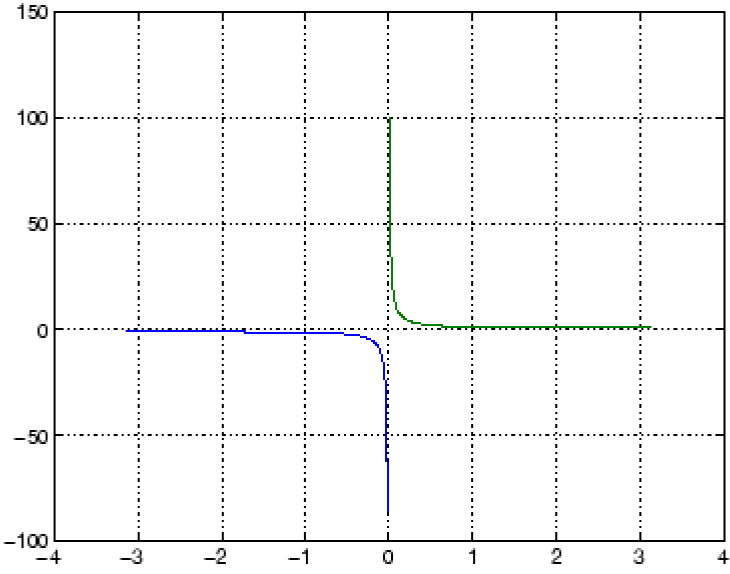
The `coth` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

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

Examples

Graph the hyperbolic cotangent over the domains $-\pi < x < 0$ and $0 < x < \pi$.

```
x1 = -pi+0.01:0.01:-0.01;
x2 = 0.01:0.01:pi-0.01;
plot(x1,coth(x1),x2,coth(x2)), grid on
```



See Also

`acoth` | `cot` | `sinh` | `cosh` | `tanh`

Purpose

Covariance matrix

Syntax

```
cov(x)
cov(X)
cov(X,Y)
cov(x) or cov(x,y)
cov(x,1)
cov(x,y,1)
```

Description

`cov(x)`, if `x` is a vector, returns the variance of `x`. For matrix input `X`, where each row is an observation, and each column is a variable, `cov(X)` is the covariance matrix. `diag(cov(X))` is a vector of variances for each column, and `sqrt(diag(cov(X)))` is a vector of standard deviations. `cov(X,Y)`, where `X` and `Y` are matrices with the same number of elements, is equivalent to `cov([X(:)  Y(:)])`.

`cov(x) or cov(x,y)` normalizes by `N` – 1, if `N` > 1, where `N` is the number of observations. This makes `cov(X)` the best unbiased estimate of the covariance matrix if the observations are from a normal distribution. For `N` = 1, cov normalizes by `N`.

`cov(x,1)` or `cov(x,y,1)` normalizes by `N` and produces the second moment matrix of the observations about their mean. `cov(X,Y,0)` is the same as `cov(X,Y)` and `cov(X,0)` is the same as `cov(X)`.

Tips

`cov` removes the mean from each column before calculating the result.

The *covariance* between two random variables is:

$$\mathrm{cov}(x_1, x_2) = E\left[(x_1 - \mu_1)^*(x_2 - \mu_2)\right]$$

where E is the mathematical expectation and $\mu_i = Ex_i$.

Examples

Consider `A = [-1 1 2 ; -2 3 1 ; 4 0 3]`. To obtain a vector of variances for each column of `A`:

```
v = diag(cov(A))'
v =
```

```
   10.3333    2.3333    1.0000
```

Compare vector v with covariance matrix C of A:

```
C = cov(A)
C =
   10.3333   -4.1667    3.0000
   -4.1667    2.3333   -1.5000
    3.0000   -1.5000    1.0000
```

The diagonal elements C(i,i) represent the variances for the columns of A. The off-diagonal elements C(i,j) represent the covariances of columns i and j.

See Also

corrcoef | mean | median | std | var | xcorr | xcov

Purpose

Sort complex numbers into complex conjugate pairs

Syntax

```
B = cplxpair(A)
B = cplxpair(A,tol)
B = cplxpair(A,[],dim)
B = cplxpair(A,tol,dim)
```

Description

`B = cplxpair(A)` sorts the elements along different dimensions of a complex array, grouping together complex conjugate pairs.

The conjugate pairs are ordered by increasing real part. Within a pair, the element with negative imaginary part comes first. The purely real values are returned following all the complex pairs. The complex conjugate pairs are forced to be exact complex conjugates. A default tolerance of `100*eps` relative to `abs(A(i))` determines which numbers are real and which elements are paired complex conjugates.

If `A` is a vector, `cplxpair(A)` returns `A` with complex conjugate pairs grouped together.

If `A` is a matrix, `cplxpair(A)` returns `A` with its columns sorted and complex conjugates paired.

If `A` is a multidimensional array, `cplxpair(A)` treats the values along the first non-singleton dimension as vectors, returning an array of sorted elements.

`B = cplxpair(A,tol)` overrides the default tolerance.

`B = cplxpair(A,[],dim)` sorts `A` along the dimension specified by scalar dim.

`B = cplxpair(A,tol,dim)` sorts `A` along the specified dimension and overrides the default tolerance.

Diagnostics

If there are an odd number of complex numbers, or if the complex numbers cannot be grouped into complex conjugate pairs within the tolerance, `cplxpair` generates the error message

```
Complex numbers can't be paired.
```

Purpose

Elapsed CPU time

Syntax

```
cputime
```

Description

`cputime` returns the total CPU time (in seconds) used by your MATLAB application from the time it was started. This number can overflow the internal representation and wrap around.

Tips

Although it is possible to measure performance using the `cputime` function, it is recommended that you use the `tic` and `toc` functions for this purpose exclusively. See Using tic and toc Versus the cputime Function in the MATLAB Programming Fundamentals documentation for more information.

Examples

The following code returns the CPU time used to run `surf(peaks(40))`.

```
t = cputime; surf(peaks(40)); e = cputime-t

e =
    0.4667
```

See Also

`clock` | `etime` | `tic` | `toc`

RandStream.create

Purpose Create random number streams

Class `RandStream`

Syntax

```
[s1,s2,...] = RandStream.create('gentype','NumStreams',n)
s = RandStream.create('gentype')
[ ... ] = RandStream.create('gentype', Name, Value,...)
```

Description

`[s1,s2,...]  = RandStream.create('gentype','NumStreams',n)` creates `n` random number streams that use the uniform pseudorandom number generator algorithm specified by `gentype`. The streams are independent in a pseudorandom sense. The streams are not necessarily independent from streams created at other times. `RandStream.list` returns all possible values for `gentype` or see "Choosing a Random Number Generator" in the MATLAB Mathematics documentation for details on generator algorithms.

Note Multiple streams are not supported by all generator types. Use either the multiplicative lagged Fibonacci generator (`mlfg6331_64`) or the combined multiple recursive generator (`mrg32k3a`) to create multiple streams.

`s = RandStream.create('gentype')` creates a single random stream. The `RandStream` constructor is a more concise alternative when you need to create a single stream.

`[ ...  ]  = RandStream.create('gentype', Name, Value,...)` allows you to specify optional Name, Value pairs to control creation of the stream. The parameters are:

NumStreams	Total number of streams of this type that will be created across sessions or labs. Default is 1.
StreamIndices	Stream indices that should be created in this call. Default is `1:N`, where `N` is the value given with the `'NumStreams'` parameter.
Seed	Nonnegative scalar integer with which to initialize all streams. Default is 0. Seeds must be an integer between 0 and $2^{32} - 1$ or `'shuffle'` to create a seed based on the current time.
NormalTransform	Transformation algorithm used by `randn(S, ...)` to generate normal pseudorandom values. Options are `'Ziggurat'`, `'Polar'`, or `'Inversion'`.
RandnAlg	`RandnAlg` has been removed. Use `NormalTransform` instead. Transformation algorithm used by `randn(S, ...)` to generate normal pseudorandom values. Options are `'Ziggurat'`, `'Polar'`, or `'Inversion'`.
CellOutput	Logical flag indicating whether or not to return the stream objects as elements of a cell array. Default is false.

Examples

Create three independent streams.

```
[s1,s2,s3] = RandStream.create('mrg32k3a','NumStreams',3);
r1 = rand(s1,100000,1);
```

```
r2 = rand(s2,100000,1);
r3 = rand(s3,100000,1);
corrcoef([r1,r2,r3])
```

Create one stream from a set of three independent streams and designate it as the global stream.

```
s2 = RandStream.create('mrg32k3a','NumStreams',3,'StreamIndices',2);
RandStream.setGlobalStream(s2);
```

See Also

`RandStream` | `RandStream.list`

Purpose

Create MATLAB class based on WSDL document

Syntax

```
createClassFromWsdl(source)
```

Description

`createClassFromWsdl(source)` creates a MATLAB class, `servicename`, based on a service name defined in `source`. The `source` argument is a string that specifies a URL, full path, or relative path to a Web Services Description Language (WSDL) document located on a server. `createClassFromWsdl` creates a class folder, `@servicename`, in the current folder. The class folder contains a method file for each Web service operation, and the display method (`display.m`) and constructor (`servicename.m`) for the class.

Examples

Get the methods from the `myWebService` WSDL document, which specifies two methods. The example does not use an actual WSDL document; therefore, you cannot run it. The example only illustrates how to use the function.

Create the class:

```
createClassFromWsdl('pathto_myWebService')
```

MATLAB creates the following in the current folder:

```
@myWebService
@myWebService/method1.m
@myWebService/method2.m
@myWebService/display.m
@myWebService/myWebService.m
```

Retrieve a student name, given the WSDL document for `TestScoreWebService`, at `http://examplestandardtests.com/scoreswebservice?WSDL`. The example does not use an actual WSDL document; therefore, you cannot run it. The example only illustrates how to use the function.

```
url = 'http://examplestandardtests.com/scoreswebservice?WSDL';
```

```
createClassFromWsdl(url);
obj = TestScoreWebService;
% Show the methods
methods(obj)
% Retrieve the first student name
students = StudentNames(obj);
students.StudentInfo(1)
```

MATLAB returns

```
StudentNameLast: 'Benjamin'
StudentNameFirst: 'Ali'
```

Display the endpoint and WSDL document location:

```
display('TestScoreWebService')
```

MATLAB returns

```
endpoint: 'http://examplestandardtests.com/scoreswebservice'
    wsdl: 'http://examplestandardtests.com/scoreswebservice?WSDL'
```

See Also

`callSoapService` | `createSoapMessage` | `parseSoapResponse` | `xmlread`

How To

- “Access Web Services That Use WSDL Documents”
- “Specify Proxy Server Settings for Connecting to the Internet”

Purpose Create copy of `inputParser` object (to be removed)

Compatibility The `createCopy` method will be removed in a future release. Use `copy` instead.

Syntax `pNew = createCopy(p)`

Description `pNew = createCopy(p)` creates a copy of `inputParser` object `p`.

Tips

- Using `createCopy` is not the same as copying by assignment, such as `pNew = p`, which creates a new handle to the same object `p`.

Input Arguments

p

Object of class `inputParser`.

Output Arguments

pNew

`inputParser` object with the same properties as object `p`.

Examples

Copy Input Parser Scheme

Create a copy of an existing `inputParser` object, and add an optional input to the copy.

```
p = inputParser;
default = 0;

addRequired(p,'first');
addOptional(p,'second',default);

pNew = createCopy(p);
addOptional(pNew,'third',default);
```

Object `p` has two inputs in the scheme, `first` and `second`. Object `pNew` has an additional input, `third`.

Parse each object.

```
input1 = 1; input2 = 2; input3 = 3;

parse(p,input1);
p.Results

ans =
     first: 1
    second: 0

parse(pNew,input1,input2,input3);
pNew.Results

ans =
     first: 1
    second: 2
     third: 3
```

See Also `inputParser` |

Purpose

Create SOAP message to send to server

Syntax

```
message = createSoapMessage(namespace,method,values,names,
   types,style)
```

Description

`message = createSoapMessage(namespace,method,values,names,types,style)` creates a SOAP message based on the values you provide for the arguments. `message` is a Java document object model (DOM). To send `message` to the Web service, use it with `callSoapService`.

Argument	Description
`namespace`	Location of the Web service in the form of a valid Uniform Resource Identifier (URI).
`method`	Name of the Web service operation you want to run.
`values`	Cell array of input you need to provide for the `method`.
`names`	Cell array of parameters for `method`.
`types`	Cell array defining the XML data types for `values`. Specifying `style` is optional; when you do not include the argument, MATLAB uses unspecified.
`style`	Style for structuring the SOAP message, either `'document'` or `'rpc'`. Specifying `style` is optional; when you do not include the argument, MATLAB uses `rpc`. Use a style supported by the service you specified in `namespace`.

Examples

This example uses `createSoapMessage` in conjunction with other SOAP functions to retrieve information about books from a library database, specifically, the author's name for a given book title.

Note The example is not based on an actual endpoint; therefore, you cannot run it. The example only illustrates how to use the SOAP functions.

```
% Create the message:
message = createSoapMessage(...
'urn:LibraryCatalog',...
'getAuthor',...
{'In the Fall'},...
{'nameToLookUp'},...
{'{http://www.w3.org/2001/XMLSchema}string'},...
'rpc');
%
% Send the message to the service and get the response:
response = callSoapService(...
'http://test/soap/services/LibraryCatalog',...
'urn:LibraryCatalog#getAuthor',...
message)
%
% Extract MATLAB data from the response
author = parseSoapResponse(response)
```

MATLAB returns:

```
author = Kate Alvin
```

where author is a char class (type).

See Also

`callSoapService` | `createClassFromWsdl` | `parseSoapResponse` | `urlread` | `xmlread`

How To

- "Access Web Services Using MATLAB SOAP Functions"

Purpose

Vector cross product

Syntax

```
C = cross(A,B)
C = cross(A,B,dim)
```

Description

`C = cross(A,B)` returns the cross product of the vectors `A` and `B`. That is, `C = A x B`. `A` and `B` must be 3-element vectors. If `A` and `B` are multidimensional arrays, `cross` returns the cross product of `A` and `B` along the first dimension of length 3.

`C = cross(A,B,dim)` where `A` and `B` are multidimensional arrays, returns the cross product of `A` and `B` in dimension `dim`. `A` and `B` must have the same size, and both `size(A,dim)` and `size(B,dim)` must be 3.

Tips

To perform a dot (scalar) product of two vectors of the same size, use `c = dot(a,b)`.

Examples

The cross and dot products of two vectors are calculated as shown:

```
a = [1 2 3];
b = [4 5 6];
c = cross(a,b)

c =
      -3     6    -3

d = dot(a,b)

d =
      32
```

See Also

`dot`

Purpose

Cosecant of argument in radians

Syntax

```
Y = csc(x)
```

Description

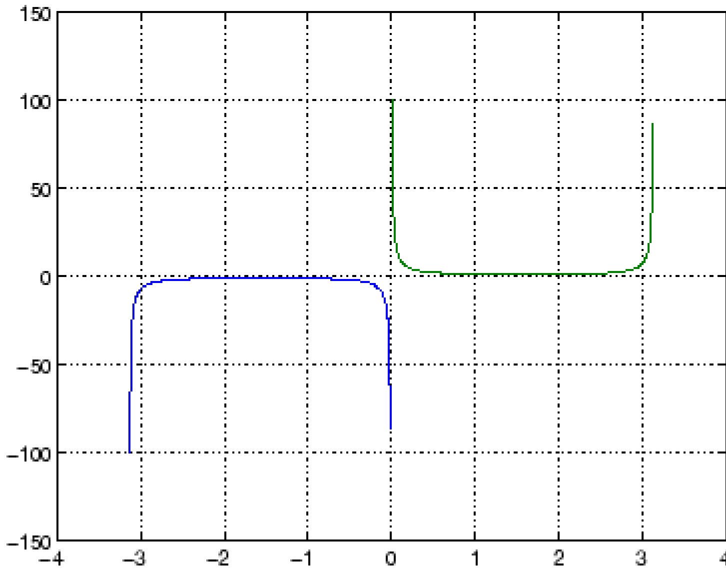
The `csc` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = csc(x)` returns the cosecant for each element of `x`.

Examples

Graph the cosecant over the domains $-\pi < x < 0$ and $0 < x < \pi$.

```
x1 = -pi+0.01:0.01:-0.01;
x2 = 0.01:0.01:pi-0.01;
plot(x1,csc(x1),x2,csc(x2)), grid on
```



See Also

`cscd` | `csch` | `acsc` | `acscd` | `acsch`

Purpose

Cosecant of argument in degrees

Syntax

```
Y = cscd(X)
```

Description

`Y = cscd(X)` returns the cosecant of the elements of `X`, which are expressed in degrees.

Input Arguments

X - Angle in degrees

scalar value | vector | matrix | N-D array

Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `cscd` operation is element-wise when `X` is nonscalar.

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

Output Arguments

Y - Cosecant of angle

scalar value | vector | matrix | N-D array

Cosecant of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as `X`.

Examples

Cosecant of 180 degrees compared to cosecant of radians

`cscd(180)` is infinite, whereas `csc(pi)` is large but finite.

```
cscd(180)

ans =

   Inf

csc(pi)

ans =
```

```
   8.1656e+15
```

Cosecant of vector of complex angles, specified in degrees

```
z = [35+i 15+2i 10+3i];
y = cscd(z)

y =

   1.7421 - 0.0434i   3.7970 - 0.4944i   5.2857 - 1.5681i
```

See Also

acscd | csc | acsc

Purpose

Hyperbolic cosecant

Syntax

```
Y = csch(x)
```

Description

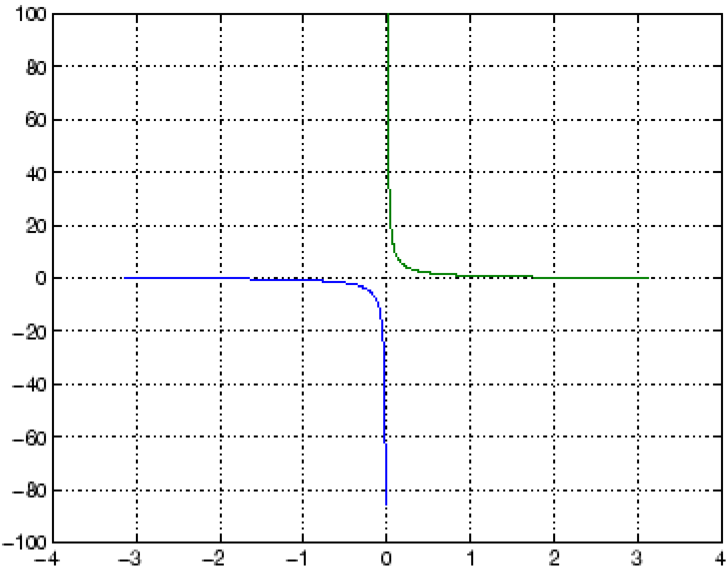
The `csch` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = csch(x)` returns the hyperbolic cosecant for each element of `x`.

Examples

Graph the hyperbolic cosecant over the domains $-\pi < x < 0$ and $0 < x < \pi$.

```
x1 = -pi+0.01:0.01:-0.01;
x2 = 0.01:0.01:pi-0.01;
plot(x1,csch(x1),x2,csch(x2)), grid on
```



See Also

`acsch` | `csc` | `sinh` | `cosh`

csvread

Purpose

Read comma-separated value file

Syntax

```
M = csvread(filename)
M = csvread(filename,row,col)
M = csvread(filename,row,col,csvRange)
```

Description

`M = csvread(filename)` reads a comma-separated value formatted file, `filename`. The file can only contain numeric values.

`M = csvread(filename,row,col)` reads data from the file starting at the specified row and column. The row and column arguments are zero based, so that `row = 0` and `col = 0` specify the first value in the file.

`M = csvread(filename,row,col,csvRange)` reads only the range specified by `csvRange`.

Input Arguments

filename - Name of file to read

string

Name of the file to read, specified as a string.

Example: `'myFile.dat'`

Data Types

`char`

row - Row of first value to read

0 (default) | positive integer

Row of the first value to read, specified as a positive integer. `row` is zero based, so that `row = 0` specifies the first row of data.

Data Types

`single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

col - Column of first value to read

0 (default) | positive integer

Column of the first value to read, specified as a positive integer. `col` is zero based, so that `col = 0` specifies the first column of data.

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

csvRange - Range of data to read

4-element vector | string

Range of the data to read, specified as a 4-element vector or a string.

- If `csvRange` is a 4-element vector, then it must have the form `[R1,C1,R2,C2]`, where (`R1,C1`) is the upper left corner of the data to be read and (`R2,C2`) is the lower right corner. The range is zero based, so that `R1 = 0` specifies the first row of data, and `C1 = 0` specifies the first column of data.
- If `csvRange` is a string, then it should be specified using spreadsheet notation, as in `csvRange = 'A1..B7'`.

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

Output Arguments

M - Numeric data

matrix

Numeric data, returned as a matrix of `double` values.

Examples

Read File Containing Comma-Separated Values

Create a file, `csvlist.dat`, that contains these comma-separated values.

```
02, 04, 06, 08
03, 06, 09, 12
05, 10, 15, 20
07, 14, 21, 28
```

Read the entire file.

```
filename = 'csvlist.dat';
M = csvread(filename)

M =

     2     4     6     8
     3     6     9    12
     5    10    15    20
     7    14    21    28
```

`csvread` returns the numeric data in `M`.

Read Data Starting at Specific Row and Column

Read data from the file, `csvlist.dat`, of the previous example, starting at zero-based row 2, column 0.

```
M = csvread('csvlist.dat',2,0)

M =

     5    10    15    20
     7    14    21    28
```

Read a Specific Range of Data

Read a specific range of data from the file, `csvlist.dat`, of the first example.

Read the matrix bounded by zero-based (1,0) and (2,2).

```
M = csvread('csvlist.dat',1,0,[1,0,2,2])

M =

     3     6     9
     5    10    15
```

Algorithms

`csvread` fills empty delimited fields with zero. When `csvread` reads data files with lines that end with a nonspace delimiter, such as a semicolon, it returns a matrix, `M`, that has an additional last column of zeros.

`csvread` imports any complex number as a whole into a complex numeric field, converting the real and imaginary parts to the specified numeric type. The table shows valid forms for a complex number.

<table>
<tr><th>Form</th><th>Example</th></tr>
<tr><td>–<real>–<imag>i|j</td><td>5.7-3.1i</td></tr>
<tr><td>–<imag>i|j</td><td>-7j</td></tr>
</table>

Embedded white-space in a complex number is invalid and is regarded as a field delimiter.

See Also

`csvwrite` | `dlmread` | `textscan` | `importdata` | `uiimport`

ctranspose

Purpose

Complex conjugate transpose

Syntax

```
b = a'
b = ctranspose(a)
```

Description

`b = a'` computes the complex conjugate transpose of matrix `a` and returns the result in `b`.

`b = ctranspose(a)` is called for the syntax `a'` (complex conjugate transpose) when `a` is an object.

See Also

`transpose`

Purpose

Write comma-separated value file

Syntax

```
csvwrite(filename,M)
csvwrite(filename,M,row,col)
```

Description

`csvwrite(filename,M)` writes matrix `M` into `filename` as comma-separated values. The `filename` input is a string enclosed in single quotes.

`csvwrite(filename,M,row,col)` writes matrix `M` into `filename` starting at the specified row and column offset. The row and column arguments are zero based, so that `row=0` and `C=0` specify the first value in the file.

Tips

- `csvwrite` terminates each line with a line feed character and no carriage return.
- `csvwrite` writes a maximum of five significant digits. If you need greater precision, use `dlmwrite` with a precision argument.
- `csvwrite` does not accept cell arrays for the input matrix `M`. To export a cell array that contains only numeric data, use `cell2mat` to convert the cell array to a numeric matrix before calling `csvwrite`. To export cell arrays with mixed alphabetic and numeric data, where each cell contains a single element, you can create an Excel spreadsheet (if your system has Excel installed) using `xlswrite`. For all other cases, you must use low-level export functions to write your data. For more information, see "Exporting a Cell Array to a Text File" in the MATLAB Data Import and Export documentation.

Examples

The following example creates a comma-separated value file from the matrix `m`.

```
m = [3 6 9 12 15; 5 10 15 20 25; ...
     7 14 21 28 35; 11 22 33 44 55];

csvwrite('csvlist.dat',m)
type csvlist.dat
```

```
3,6,9,12,15
5,10,15,20,25
7,14,21,28,35
11,22,33,44,55
```

The next example writes the matrix to the file, starting at a column offset of 2.

```
csvwrite('csvlist.dat',m,0,2)
type csvlist.dat

,,3,6,9,12,15
,,5,10,15,20,25
,,7,14,21,28,35
,,11,22,33,44,55
```

See Also

csvread | dlmwrite | xlswrite | importdata | uiimport

Purpose

Cumulative product

Syntax

```
B = cumprod(A)
B = cumprod(A,dim)
```

Description

`B = cumprod(A)` returns an array the same size as the array `A` containing the cumulative product.

- If `A` is a vector, then `cumprod(A)` returns a vector containing the cumulative product of the elements of `A`.
- If `A` is a matrix, then `cumprod(A)` returns a matrix containing the cumulative products for each column of `A`.
- If `A` is a multidimensional array, then `cumprod(A)` acts along the first nonsingleton dimension.

`B = cumprod(A,dim)` returns the cumulative product along dimension `dim`. For example, if `A` is a matrix, then `cumprod(A,2)` returns the cumulative product of each row.

Input Arguments

A - Input array

vector | matrix | multidimensional array

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

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

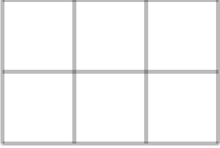
dim - Dimension to operate along

positive integer scalar

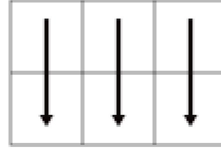
Dimension to operate along, specified as a positive integer scalar. If no value is specified, the default is the first nonsingleton dimension.

Consider a two-dimensional input array, `A`.

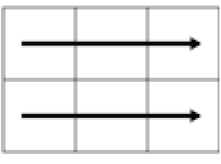
- `cumprod(A,1)` works along the rows of `A` and returns the cumulative product of each column.
- `cumprod(A,2)` works along the columns of `A` and returns the cumulative product of each row.



A



cumprod(A,1)



cumprod(A,2)

`cumprod` returns `A` if `dim` is greater than `ndims(A)`.

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

Output Arguments

B - Cumulative product array

vector | matrix | multidimensional array

Cumulative product array, returned as a vector, matrix, or multidimensional array of the same size as the input array `A`.

The class of `B` is the same as the class of `A` except if `A` is `logical`, in which case `B` is `double`.

Definitions

First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to `1`.

For example:

- If `X` is a 1-by-n row vector, then the second dimension is the first nonsingleton dimension of `X`.
- If `X` is a 1-by-0-by-n empty array, then the second dimension is the first nonsingleton dimension of `X`.

- If `X` is a 1-by-1-by-3 array, then the third dimension is the first nonsingleton dimension of `X`.

Examples

Cumulative Product of Vector

Find the cumulative product of the integers from `1` to `5`.

```
A = [1:5];
B = cumprod(A)

B =

     1     2     6    24   120
```

`B(2)` is the product of `A(1)` and `A(2)`, while `B(5)` is the product of elements `A(1)` through `A(5)`.

Cumulative Product of Each Column in Matrix

Define a 3-by-3 matrix whose elements correspond to their linear indices.

```
A = [1 4 7; 2 5 8; 3 6 9]

A =

     1     4     7
     2     5     8
     3     6     9
```

Find the cumulative product of the columns of `A`.

```
B = cumprod(A)

B =

     1     4     7
     2    20    56
     6   120   504
```

B(5) is the product of A(4) and A(5), while B(9) is the product of A(7) , A(8), and A(9).

Cumulative Product of Each Row in Matrix

Define a 2-by-3 matrix whose elements correspond to their linear indices.

```
A = [1 3 5; 2 4 6]
```

```
A =

     1     3     5
     2     4     6
```

Find the cumulative product of the rows of A.

```
B = cumprod(A,2)
```

```
B =

     1     3    15
     2     8    48
```

B(3) is the product of A(1) and A(3), while B(5) is the product of A(1), A(3), and A(5) .

Logical Input with Double Output

Create an array of logical values.

```
A = [true false true; true true false]
```

```
A =

     1     0     1
     1     1     0
```

Find the cumulative product of the rows of A.

```
B = cumprod(A,2)

B =

     1     0     0
     1     1     0
```

The output is `double`.

```
class(B)

ans =

double
```

See Also

`cumsum` | `prod` | `sum`

Purpose

Cumulative sum

Syntax

```
B = cumsum(A)
B = cumsum(A,dim)
```

Description

`B = cumsum(A)` returns an array `A` containing the cumulative sum.

- If `A` is a vector, then `cumsum(A)` returns a vector containing the cumulative sum of the elements of `A`.
- If `A` is a matrix, then `cumsum(A)` returns a matrix containing the cumulative sums for each column of `A`.
- If `A` is a multidimensional array, then `cumsum(A)` acts along the first nonsingleton dimension.

`B = cumsum(A,dim)` returns the cumulative sum of the elements along dimension `dim`. For example, if `A` is a matrix, then `cumsum(A,2)` returns the cumulative sum of each row.

Input Arguments

A - Input array

vector | matrix | multidimensional array

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

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

dim - Dimension to operate along

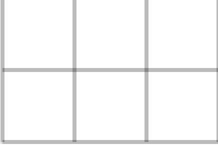
positive integer scalar

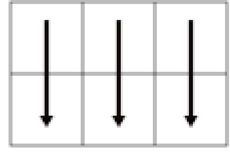
Dimension to operate along, specified as a positive integer scalar. If no value is specified, the default is the first nonsingleton dimension.

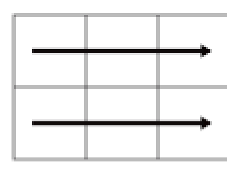
Consider a two-dimensional input array, `A`:

- `cumsum(A,1)` works along the rows of `A` and returns the cumulative sum of each column.

- `cumsum(A,2)` works along the columns of `A` and returns the cumulative sum of each row.







`cumsum` returns `A` if `dim` is greater than `ndims(A)`.

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

Output Arguments

B - Cumulative sum array

vector | matrix | multidimensional array

Cumulative sum array, returned as a vector, matrix, or multidimensional array of the same size as the input array `A`.

The class of `B` is the same as the class of `A` except if `A` is `logical`, in which case `B` is `double`.

Definitions

First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to `1`.

For example:

- If `X` is a 1-by-n row vector, then the second dimension is the first nonsingleton dimension of `X`.
- If `X` is a 1-by-0-by-n empty array, then the second dimension is the first nonsingleton dimension of `X`.
- If `X` is a 1-by-1-by-3 array, then the third dimension is the first nonsingleton dimension of `X`.

Examples

Cumulative Sum of Vector

Find the cumulative sum of the integers from `1` to `5`.

```
A = [1:5];
B = cumsum(A)
```

```
B =

     1     3     6    10    15
```

`B(2)` is the sum of `A(1)` and `A(2)`, while `B(5)` is the sum of elements `A(1)` through `A(5)`.

Cumulative Sum of Each Column in Matrix

Define a 3-by-3 matrix whose elements correspond to their linear indices.

```
A = [1 4 7; 2 5 8; 3 6 9]
```

```
A =

     1     4     7
     2     5     8
     3     6     9
```

Find the cumulative sum of the columns of `A`.

```
B = cumsum(A)
```

```
B =

     1     4     7
     3     9    15
     6    15    24
```

`B(5)` is the sum of `A(4)` and `A(5)`, while `B(9)` is the sum of `A(7)` , `A(8)`, and `A(9)`.

Cumulative Sum of Each Row in Matrix

Define a 2-by-3 matrix whose elements correspond to their linear indices.

```
A = [1 3 5; 2 4 6]
```

```
A =

     1     3     5
     2     4     6
```

Find the cumulative sum of the rows of A.

```
B = cumsum(A,2)
```

```
B =

     1     4     9
     2     6    12
```

`B(3)` is the sum of `A(1)` and `A(3)`, while `B(5)` is the sum of `A(1)`, `A(3)`, and `A(5)` .

Logical Input with Double Output

Create an array of logical values.

```
A = [true false true; true true false]
```

```
A =

     1     0     1
     1     1     0
```

Find the cumulative sum of the rows of A.

```
B = cumsum(A,2)
```

```
B =

     1     1     2
     1     2     2
```

The output is `double`.

```
class(B)
```

```
ans =

double
```

See Also

`cumprod` | `prod` | `sum` | `diff`

Purpose

Cumulative trapezoidal numerical integration

Syntax

```
Z = cumtrapz(Y)
Z = cumtrapz(X,Y)
Z = cumtrapz(X,Y,dim) or cumtrapz(Y,dim)
```

Description

`Z = cumtrapz(Y)` computes an approximation of the cumulative integral of `Y` via the trapezoidal method with unit spacing. To compute the integral with other than unit spacing, multiply `Z` by the spacing increment. Input `Y` can be complex.

For vectors, `cumtrapz(Y)` is a vector containing the cumulative integral of `Y`.

For matrices, `cumtrapz(Y)` is a matrix the same size as `Y` with the cumulative integral over each column.

For multidimensional arrays, `cumtrapz(Y)` works across the first nonsingleton dimension.

`Z = cumtrapz(X,Y)` computes the cumulative integral of `Y` with respect to `X` using trapezoidal integration. `X` and `Y` must be vectors of the same length, or `X` must be a column vector and `Y` an array whose first nonsingleton dimension is `length(X)`. `cumtrapz` operates across this dimension. Inputs `X` and `Y` can be complex.

If `X` is a column vector and `Y` an array whose first nonsingleton dimension is `length(X)`, `cumtrapz(X,Y)` operates across this dimension.

`Z = cumtrapz(X,Y,dim) or cumtrapz(Y,dim)` integrates across the dimension of `Y` specified by scalar `dim`. The length of `X` must be the same as `size(Y,dim)`.

Examples

Example 1

```
Y = [0 1 2; 3 4 5];

cumtrapz(Y,1)
ans =
0         0         0
```

```
    1.5000   2.5000    3.5000

cumtrapz(Y,2)
ans =
0    0.5000    2.0000
        0    3.5000    8.0000
```

Example 2

This example uses two complex inputs:

```
z = exp(1i*pi*(0:100)/100);

ct = cumtrapz(z,1./z);
ct(end)
ans =
   0.0000 + 3.1411i
```

See Also

cumsum | trapz

curl

Purpose

Compute curl and angular velocity of vector field

Syntax

```
[curlx,curly,curlz,cav] = curl(X,Y,Z,U,V,W)
[curlx,curly,curlz,cav] = curl(U,V,W)
[curlz,cav]= curl(X,Y,U,V)
[curlz,cav]= curl(U,V)
[curlx,curly,curlz] = curl(...), [curlx,curly] = curl(...)
cav = curl(...)
```

Description

`[curlx,curly,curlz,cav] = curl(X,Y,Z,U,V,W)` computes the curl (`curlx`, `curly`, `curlz`) and angular velocity (`cav`) perpendicular to the flow (in radians per time unit) of a 3-D vector field `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`.

`[curlx,curly,curlz,cav] = curl(U,V,W)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`[curlz,cav]= curl(X,Y,U,V)` computes the curl `z`-component and the angular velocity perpendicular to `z` (in radians per time unit) of a 2-D vector field `U`, and `V`.

The arrays `X` and `Y`, which define the coordinates for `U` and `V`, must be monotonic, but do not need to be uniformly spaced. `X` and `Y` must have the same number of elements, as if produced by `meshgrid`.

`[curlz,cav]= curl(U,V)` assumes `X` and `Y` are determined by the expression

```
[X Y] = meshgrid(1:n,1:m)
```

where `[m,n] = size(U)`.

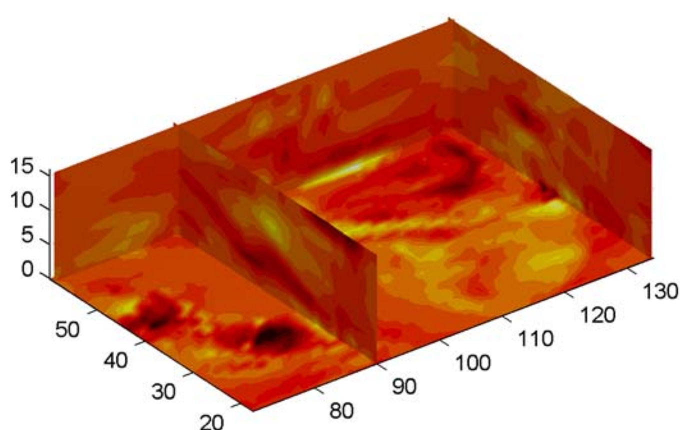
`[curlx,curly,curlz] = curl(...)`, `[curlx,curly] = curl(...)` returns only the curl.

`cav = curl(...)` returns only the curl angular velocity.

Examples

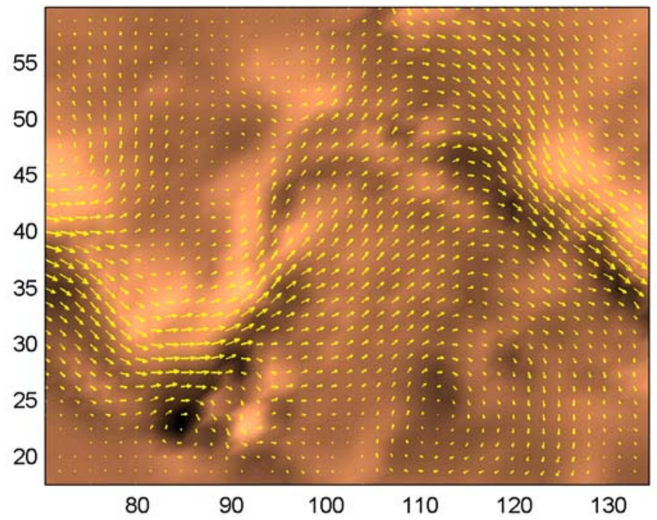
This example uses colored slice planes to display the curl angular velocity at specified locations in the vector field.

```
figure
load wind
cav = curl(x,y,z,u,v,w);
h = slice(x,y,z,cav,[90 134],59,0);
shading interp
daspect([1 1 1]);
axis tight
colormap hot(16)
camlight
set([h(1),h(2)],'ambientstrength',.6)
```



This example views the curl angular velocity in one plane of the volume and plots the velocity vectors (quiver) in the same plane.

```
load wind
k = 4;
x = x(:,:,k); y = y(:,:,k); u = u(:,:,k); v = v(:,:,k);
cav = curl(x,y,u,v);
pcolor(x,y,cav); shading interp
hold on;
quiver(x,y,u,v,'y')
hold off
colormap copper
```



See Also

streamribbon | divergence

How To

- “Displaying Curl with Stream Ribbons”

Tiff.currentDirectory

Purpose

Index of current IFD

Syntax

```
dirNum = tiffobj.currentDirectory()
```

Description

`dirNum = tiffobj.currentDirectory()` returns the index of the current image file directory (IFD). Index values are one-based. Use this index value with the `setDirectory` member function.

Examples

Open a `Tiff` object and determine which IFD is the current IFD. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path:

```
t = Tiff('myfile.tif', 'r');
dnum = t.currentDirectory();
```

References

This method corresponds to the `TIFFCurrentDirectory` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTiff - TIFF Library and Utilities`.

See Also

`Tiff.setDirectory`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

Purpose

Allow custom source control system (UNIX platforms)

Syntax

`customverctrl`

Description

`customverctrl` function is for customers who want to integrate a source control system that is not supported for use with MATLAB software. When using this function, conform to the structure of one of the supported version control systems, for example, RCS. For examples, see the files `clearcase.m`, `cvs.m`, `pvcs.m`, and `rcs.m` in *`matlabroot`*`\toolbox\matlab\verctrl`.

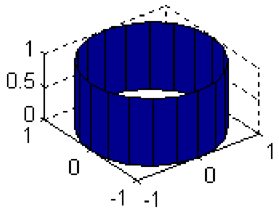
See Also

`checkin` | `checkout` | `cmopts` | `undocheckout` | `verctrl`

cylinder

Purpose

Generate cylinder



Syntax

```
[X,Y,Z] = cylinder
[X,Y,Z] = cylinder(r)
[X,Y,Z] = cylinder(r,n)
cylinder(axes_handle,...)
cylinder(...)
```

Description

`cylinder` generates *x*-, *y*-, and *z*-coordinates of a unit cylinder. You can draw the cylindrical object using `surf` or `mesh`, or draw it immediately by not providing output arguments.

`[X,Y,Z] = cylinder` returns the *x*-, *y*-, and *z*-coordinates of a cylinder with a radius equal to `1`. The cylinder has 20 equally spaced points around its circumference.

`[X,Y,Z] = cylinder(r)` returns the *x*-, *y*-, and *z*-coordinates of a cylinder using `r` to define a profile curve. `cylinder` treats each element in `r` as a radius at equally spaced heights along the unit height of the cylinder. The cylinder has 20 equally spaced points around its circumference.

`[X,Y,Z] = cylinder(r,n)` returns the *x*-, *y*-, and *z*-coordinates of a cylinder based on the profile curve defined by vector `r`. The cylinder has `n` equally spaced points around its circumference.

`cylinder(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`cylinder(...)`, with no output arguments, plots the cylinder using `surf`.

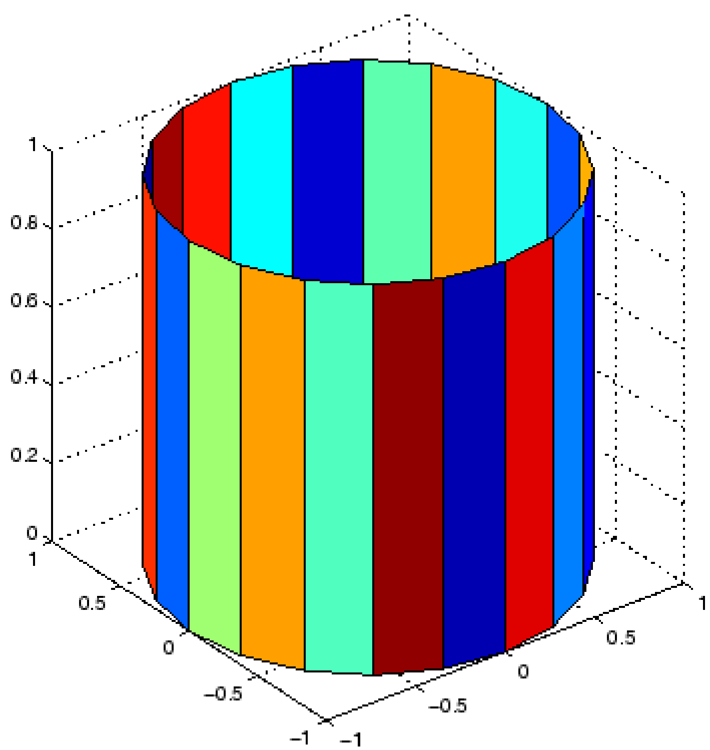
Tips

`cylinder` treats its first argument as a profile curve. The resulting surface graphics object is generated by rotating the curve about the *x*-axis, and then aligning it with the *z*-axis.

Examples

Create a cylinder with randomly colored faces.

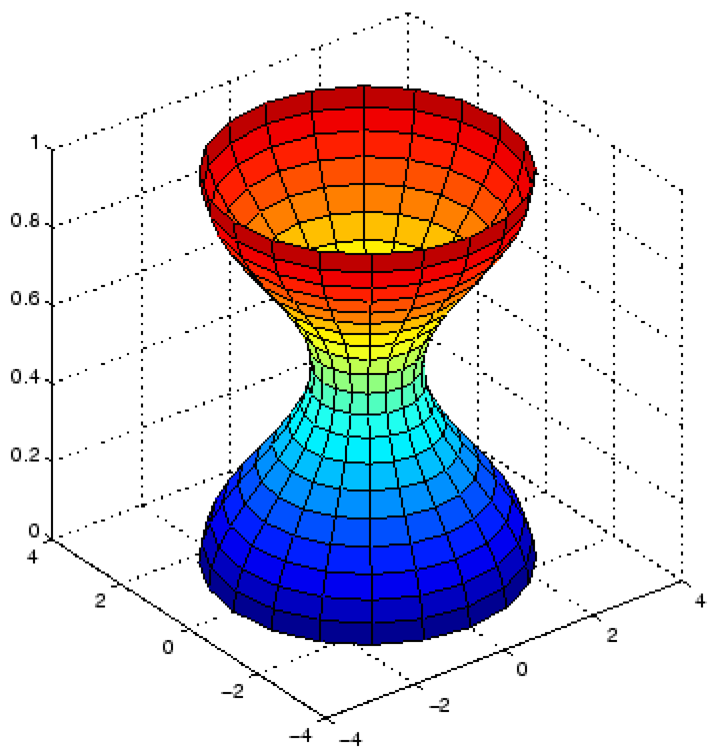
```
cylinder
axis square
h = findobj('Type','surface');
set(h,'CData',rand(size(get(h,'CData'))))
```



Generate a cylinder defined by the profile function `2+sin(t)`.

```
t = 0:pi/10:2*pi;
```

```
[X,Y,Z] = cylinder(2+cos(t));
surf(X,Y,Z)
axis square
```



See Also

sphere | surf

Purpose

Read Data Acquisition Toolbox (.daq) file

Syntax

```
data = daqread('filename')
[data, time] = daqread(...)
[data, time, abstime] = daqread(...)
[data, time, abstime, events] = daqread(...)
[data, time, abstime, events, daqinfo] = daqread(...)
data = daqread(...,'Param1', Val1,...)
daqinfo = daqread('filename','info')
```

Description

`data = daqread('filename')` reads all the data from the Data Acquisition Toolbox™ (`.daq`) file specified by `filename`. `daqread` returns `data`, an *m*-by-*n* data matrix, where *m* is the number of samples and *n* is the number of channels. If `data` includes data from multiple triggers, the data from each trigger is separated by a `NaN`. If you set the `OutputFormat` property to `tscollection`, `daqread` returns a time series collection object. See below for more information.

`[data, time] = daqread(...)` returns time/value pairs. `time` is an *m*-by-1 vector, the same length as `data`, that contains the relative time for each sample. Relative time is measured with respect to the first trigger that occurs.

`[data, time, abstime] = daqread(...)` returns the absolute time of the first trigger. `abstime` is returned as a `clock` vector.

`[data, time, abstime, events] = daqread(...)` returns a log of events. `events` is a structure containing event information. If you specify either the`Samples`, `Time`, or `Triggers` parameters (see below), the events structure contains only the specified events.

`[data, time, abstime, events, daqinfo] = daqread(...)` returns a structure, `daqinfo`, that contains two fields: `ObjInfo` and `HwInfo`. `ObjInfo` is a structure containing property name/property value pairs and `HwInfo` is a structure containing hardware information. The entire event log is returned to `daqinfo.ObjInfo.EventLog`.

`data = daqread(...,'Param1', Val1,...)` specifies the amount of data returned and the format of the data, using the following parameters.

Parameter	Description
`Samples`	Specify the sample range.
`Time`	Specify the relative time range.
`Triggers`	Specify the trigger range.
`Channels`	Specify the channel range. Channel names can be specified as a cell array.
`DataFormat`	Specify the data format as `doubles` (default) or `native`.
`TimeFormat`	Specify the time format as `vector` (default) or `matrix`.
`OutputFormat`	Specify the output format as `matrix` (the default) or `tscollection`. When you specify `tscollection`, `daqread` only returns `data`.

The `Samples`, `Time`, and `Triggers` properties are mutually exclusive; that is, either `Samples`, `Triggers` or `Time` can be defined at once.

`daqinfo = daqread('filename','`**`info`**`')` returns metadata from the file in the `daqinfo` structure, without incurring the overhead of reading the data from the file as well. The `daqinfo` structure contains two fields:

`daqinfo.ObjInfo`
: a structure containing parameter/value pairs for the data acquisition object used to create the file, `filename`. Note: The `UserData` property value is not restored.

`daqinfo.HwInfo`
: a structure containing hardware information. The entire event log is returned to `daqinfo.ObjInfo.EventLog`.

Tips

More About .daq Files

- The format used by `daqread` to return data, relative time, absolute time, and event information is identical to the format used by the `getdata` function that is part of Data Acquisition Toolbox. For more information, see the Data Acquisition Toolbox documentation.
- If data from multiple triggers is read, then the size of the resulting data array is increased by the number of triggers issued because each trigger is separated by a `NaN`.
- `ObjInfo.EventLog` always contains the entire event log regardless of the value specified by `Samples`, `Time`, or `Triggers`.
- The `UserData` property value is not restored when you return device object (`ObjInfo`) information.
- When reading a `.daq` file, the `daqread` function does not return property values that were specified as a cell array.
- Data Acquisition Toolbox (`.daq`) files are created by specifying a value for the `LogFileName` property (or accepting the default value), and configuring the `LoggingMode` property to `Disk` or `Disk&Memory`.

More About Time Series Collection Object Returned

When `OutputFormat` is set to `tscollection`, `daqread` returns a time series collection object. This times series collection object contains an absolute time series object for each channel in the file. The following describes how `daqread` sets some of the properties of the times series collection object and the time series objects.

- The `time` property of the time series collection object is set to the value of the `InitialTriggerTime` property specified in the file.
- The `name` property of each time series object is set to the value of the `Name` property of a channel in the file. If this name cannot be used as a time series object name, `daqread` sets the name to `'Channel'` with the `HwChannel` property of the channel appended.
- The value of the `Units` property of the time series object depends on the value of the `DataFormat` parameter. If the `DataFormat` parameter

is set to 'double', daqread sets the DataInfo property of each time series object in the collection to the value of the Units property of the corresponding channel in the file. If the DataFormat parameter is set to 'native', daqread sets the Units property to 'native'. See the Data Acquisition Toolbox documentation for more information on these properties.

- Each time series object will have tsdata.event objects attached corresponding to the log of events associated with the channel.

If daqread returns data from multiple triggers, the data from each trigger is separated by a NaN in the time series data. This increases the length of data and time vectors in the time series object by the number of triggers.

Examples

Use Data Acquisition Toolbox to acquire data. The analog input object, ai, acquires one second of data for four channels, and saves the data to the output file data.daq.

```
ai = analoginput('nidaq','Dev1');
chans = addchannel(ai,0:3);
set(ai,'SampleRate',1000)
ActualRate = get(ai,'SampleRate');
set(ai,'SamplesPerTrigger, ActualRate)
set(ai,'LoggingMode','Disk&Memory')
set(ai,'LogFileName','data.daq')
start(ai)
```

After the data has been collected and saved to a disk file, you can retrieve the data and other acquisition-related information using daqread. To read all the sample-time pairs from data.daq:

```
[data,time] = daqread('data.daq');
```

To read samples 500 to 1000 for all channels from data.daq:

```
data = daqread('data.daq','Samples',[500 1000]);
```

To read only samples 1000 to 2000 of channel indices 2, 4 and 7 in native format from the file, `data.daq`:

```
data = daqread('data.daq', 'Samples', [1000 2000],...
               'Channels', [2 4 7], 'DataFormat', 'native');
```

To read only the data which represents the first and second triggers on all channels from the file, `data.daq`:

```
[data, time] = daqread('data.daq', 'Triggers', [1 2]);
```

To obtain the channel property information from `data.daq`:

```
daqinfo = daqread('data.daq','info');
chaninfo = daqinfo.ObjInfo.Channel;
```

To obtain a list of event types and event data contained by `data.daq`:

```
daqinfo = daqread('data.daq','info');
events = daqinfo.ObjInfo.EventLog;
event_type = {events.Type};
event_data = {events.Data};
```

To read all the data from the file `data.daq` and return it as a time series collection object:

```
data = daqread('data.daq','OutputFormat','tscollection');
```

See Also

`timeseries` | `tscollection`

daspect

Purpose

Set or query axes data aspect ratio

Syntax

```
daspect
daspect([aspect_ratio])
daspect('mode')
daspect('auto')
daspect('manual')
daspect(axes_handle,...)
```

Description

The data aspect ratio determines the relative scaling of the data units along the *x*-, *y*-, and *z*-axes.

`daspect` with no arguments returns the data aspect ratio of the current axes.

`daspect([aspect_ratio])` sets the data aspect ratio in the current axes to the specified value. Specify the aspect ratio as three relative values representing the ratio of the *x*-, *y*-, and *z*-axis scaling (e.g., `[1 1 3]` means one unit in *x* is equal in length to one unit in *y* and three units in *z*).

`daspect('mode')` returns the current value of the data aspect ratio mode, which can be either `auto` (the default) or `manual`. See TipsRemarks.

`daspect('auto')` sets the data aspect ratio mode to `auto`.

`daspect('manual')` sets the data aspect ratio mode to `manual`.

`daspect(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, `daspect` operates on the current axes.

Tips

`daspect` sets or queries values of the axes object `DataAspectRatio` and `DataAspectRatioMode` properties.

When the data aspect ratio mode is `auto`, the data aspect ratio adjusts so that each axis spans the space available in the figure window. If you are displaying a representation of a real-life object, you should set the data aspect ratio to `[1 1 1]` to produce the correct proportions.

Setting a value for data aspect ratio or setting the data aspect ratio mode to `manual` disables the MATLAB stretch-to-fill feature (stretching of the axes to fit the window). This means setting the data aspect ratio to a value, including its current value,

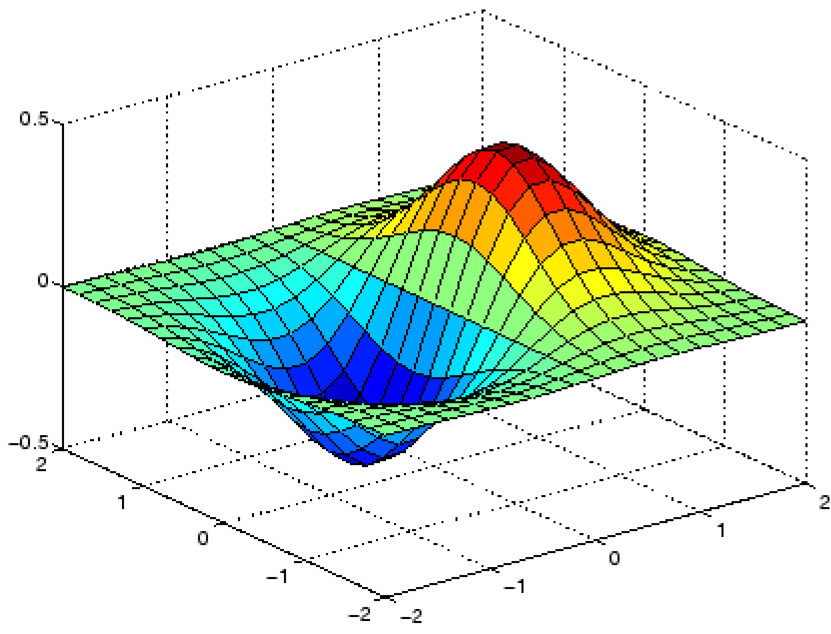
```
daspect(daspect)
```

can cause a change in the way the graphs look. See the Remarks section of the `axes` description for more information.

Examples

The following surface plot of the function $z = xe^{(-x^2 - y^2)}$ is useful to illustrate the data aspect ratio. First plot the function over the range $-2 \le x \le 2$, $-2 \le y \le 2$,

```
[x,y] = meshgrid([-2:.2:2]);
z = x.*exp(-x.^2 - y.^2);
surf(x,y,z)
```

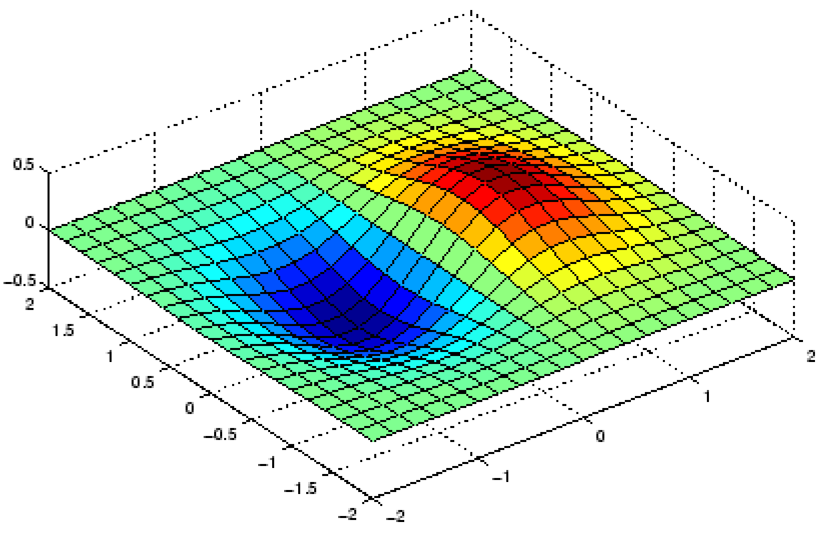


Querying the data aspect ratio shows how the surface is drawn.

```
daspect
ans =
     4   4   1
```

Setting the data aspect ratio to [1 1 1] produces a surface plot with equal scaling along each axis.

```
daspect([1 1 1])
```



See Also

axis | pbaspect | xlim | ylim | zlim

How To

- DataAspectRatio
- PlotBoxAspectRatio
- XLim
- YLim
- ZLim

- “Understanding Axes Aspect Ratio”

datacursormode

Purpose

Enable, disable, and manage interactive data cursor mode

Syntax

```
datacursormode on
datacursormode off
datacursormode
datacursormode toggle
datacursormode(figure_handle)
dcm_obj = datacursormode(figure_handle)
```

Description

`datacursormode on` enables data cursor mode on the current figure.

`datacursormode off` disables data cursor mode on the current figure.

`datacursormode` or `datacursormode toggle` toggles data cursor mode in the current figure.

`datacursormode(figure_handle)` enables or disables data cursor mode on the specified figure.

`dcm_obj = datacursormode(figure_handle)` returns the data cursor mode object for the figure. The object enables you to customize the data cursor. For more information on data cursor mode objects, see "Output Arguments" on page 1-1165. You cannot change the state of data cursor mode in a call to `datacursormode` that returns a mode object.

A *data cursor* is a small black square with a white border that you interactively position on a graph in data cursor mode. When you click a graphic object such as a line on a graph, a *data tip* appears. Data tips are small text boxes or windows that float within an axes that display data values at data cursor locations. The default style is a text box. Data tips list *x*-, *y*- and (where appropriate) *z*-values for one data point at a time. See "Examples" on page 1-1169 for an illustration of these two styles.

Tips

- Most types of graphs and 3-D plots support data cursor mode, but several do not (`pareto`, for example).
- Polar plots support data tips, but display Cartesian rather than polar coordinates on them.

- Histograms created with `hist` display specialized data tips that itemize the observation counts, lower and upper limits and center point for histogram bins.
- You place data tips only by clicking data objects on graphs. You cannot place them programmatically (by executing code to position a data cursor).
- When `DisplayStyle` is `datatip`, you can place multiple data tips on a graph. When `DisplayStyle` is `window`, it reports only the most recent data tip.
- `datacursormode off` exits data cursor mode but does not remove displayed data tips. However, if the `DisplayStyle` is `window`, the data tip window goes away.

Input Arguments

figure_handle

Optional handle of figure window

Default: The current figure

state

`''`, `'toggle'`, `'on'`, or `'off'`

Default: `'toggle'`

Output Arguments

dcm_obj

Use the object returned by `datacursormode` to control aspects of data cursor behavior. You can use the `set` and `get` commands to set and query object property values. You can customize how data cursor mode presents information by coding callback functions for these objects.

Parameter Name/Value Pairs for Data Cursor Mode Objects

The following parameters apply to objects returned by calls to `datacursormode`, not to the function itself.

'DisplayStyle'

`datatip | window`

Determines how the data cursor displays.

- `datatip` displays data cursor information in a small yellow text box attached to a black square marker at a data point you interactively select.
- `window` displays data cursor information for the data point you interactively select in a floating window within the figure.

Default: `datatip`

'Enable'

`on | off`

Specifies whether data cursor mode is currently enabled for the figure.

Default: `off`

'Figure'

handle

Handle of the figure associated with the data cursor mode object.

'SnapToDataVertex'

`on | off`

Specifies whether the data cursor snaps to the nearest data value or is located at the actual pointer position.

Default: `on`

'UpdateFcn'

function handle

Reference to a function that formats the text appearing in the data cursor. You can supply your own function to customize data tip display. Your function must include at least two arguments. The first argument is unused, and can be a variable name or tilde (~). The second argument passes the data cursor event object to your update function. The event object encapsulates the state of the data cursor. The following function definition illustrates the update function:

```
function output_txt = myfunction(~,event_obj)
% ~            Currently not used (empty)
% event_obj    Object containing event data structure
% output_txt   Data cursor text (string or cell array of strings)
```

`event_obj` is an object that has the following properties.

`Target`	Handle of the object the data cursor is referencing (the object which you click, for example, a line or a bar from a series)
`Position`	An array specifying the *x*, *y* (and *z* for 3-D graphs) coordinates of the cursor

You can query these properties within your function. For example,

```
pos = get(event_obj,'Position');
```

returns the coordinates of the cursor. Another way of accessing that data is to obtain the struct and query its `Position` field:

```
eventdata = get(event_obj);
pos = eventdata.Position;
```

You can also obtain the position directly from the object:

```
pos = event_obj.Position;
```

You can redefine the data cursor `Updatefcn` at run time. For example:

```
set(dcm_obj,'UpdateFcn',@myupdatefcn)
```

applies the function `myupdatefcn` to the current data tip or tips. When you set an update function in this way, the function must be on the MATLAB path. If instead you select the data cursor mode context menu item **Select text update function**, you can interactively select a function that is not on the path.

Do not redefine figure window callbacks, such as `ButtonDownFcn`, `KeyPpressFcn`, or `CloseRequestFcn` *while in data cursor mode.* If you attempt to change any *figure* callbacks when you are in an interactive mode, you receive a warning and the attempt fails. MATLAB interactive modes are:

- `brush`
- `datacursormode`
- `pan`
- `rotate3d`
- `zoom`

This restriction does not apply to changing the figure `WindowButtonMotionFcn` callback or `uicontrol` callbacks.

Querying Data Cursor Mode

Use the `getCursorInfo` function to query the data cursor mode object (`dcm_obj` in the update function syntax) to obtain information about the data cursor. For example,

```
info_struct = getCursorInfo(dcm_obj);
```

returns a vector of structures, one for each data cursor on the graph. Each structure has the following fields.

`Target`	The handle of the graphics object containing the data point
`Position`	An array specifying the x, y, (and z) coordinates of the cursor

Line and lineseries objects have an additional field.

`DataIndex`	A scalar index into the data arrays that correspond to the nearest data point. The value is the same for each array.

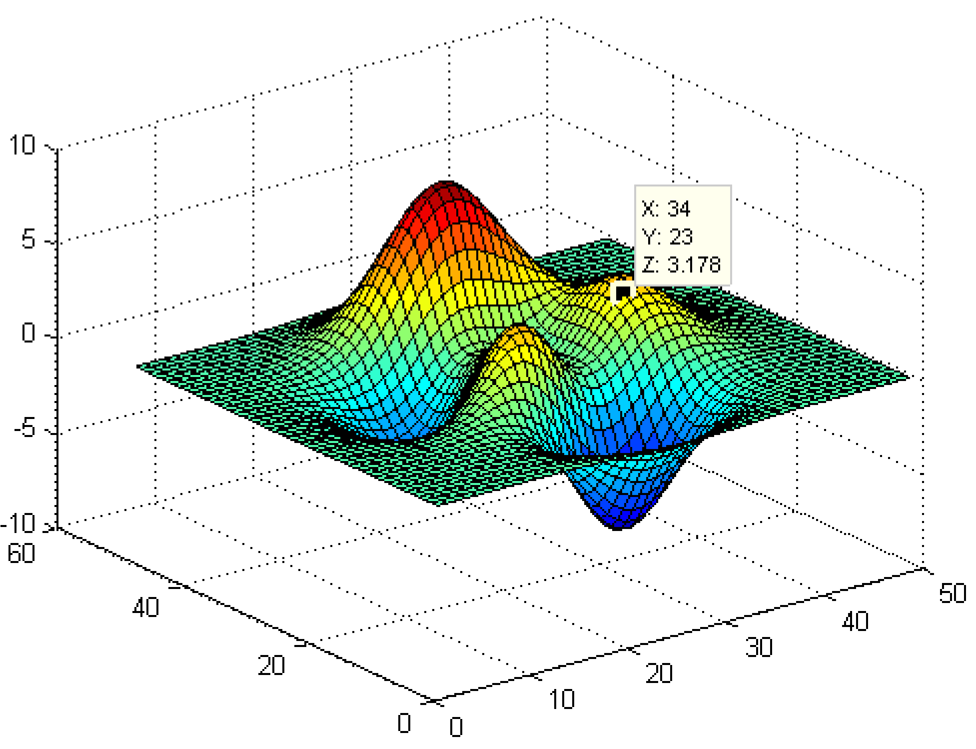
See "Output Arguments" on page 1-1165 for more details on data cursor mode objects.

Examples

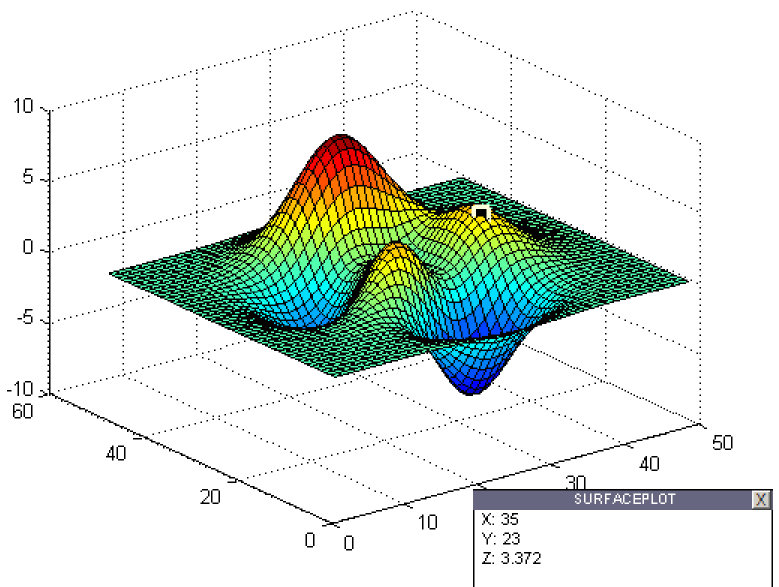
This example creates a plot and enables data cursor mode from the command line.

```
surf(peaks)
datacursormode on
% Click mouse on surface to display data cursor
```

Selecting a point on the surface opens a data tip displaying its *x*-, *y*-, and z-coordinates.



You change the data tip display style to be a window instead of a text box using the **Tools > Options > Display cursor in window** , or use the context menu **Display Style > Window inside figure** to view the data tip in a floating window that you can move around inside the axes.



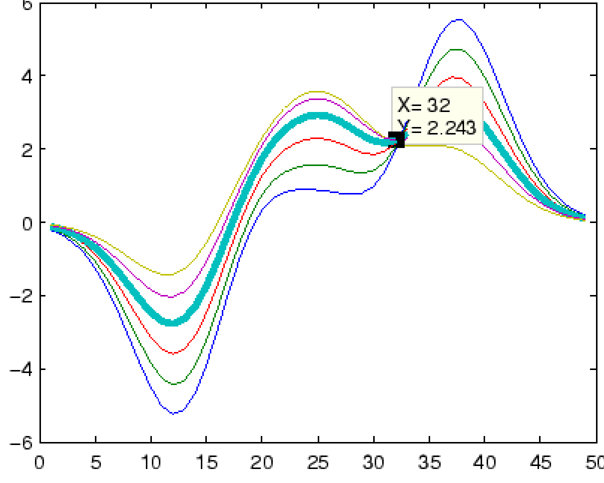
You can position multiple text box data tips on the same graph, the window style of data tip displays only one value at a time. For more information on interacting with data cursors, including point selection options and exporting data tips to the workspace, see "Data Cursor — Displaying Data Values Interactively".

This example enables data cursor mode on the current figure and sets data cursor mode options. The following statements

- Create a graph
- Toggle data cursor mode to on
- Obtain the data cursor mode object, specify data tip options, and get the handle of the line the data tip occupies:

```
fig = figure;
z = peaks;
plot(z(:,30:35))
```

```
dcm_obj = datacursormode(fig);
set(dcm_obj,'DisplayStyle','datatip',...
    'SnapToDataVertex','off','Enable','on')

disp('Click on a line to display a data tip, then press Return.')
pause                             % Wait while the user does this.

c_info = getCursorInfo(dcm_obj);
set(c_info.Target,'LineWidth',2)  % Make selected line wider
```



This example shows you how to customize the text that the data cursor displays. For example, you can replace the text displayed in the data tip and data window (`x:` and `y:`) with `Time:` and `Amplitude:` by creating a simple update function.

Save the following functions in your current directory or any writable directory on the MATLAB path before running them. As they are functions, you cannot highlight them and then evaluate the selection to make them work.

Save this code as `doc_datacursormode.m`:

```
function doc_datacursormode
% Plots graph and sets up a custom data tip update function
fig = figure;
a = -16; t = 0:60;
plot(t,sin(a*t))
dcm_obj = datacursormode(fig);
set(dcm_obj,'UpdateFcn',@myupdatefcn)
```

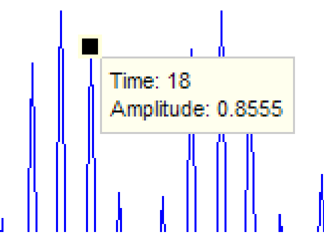
Save the following code as `myupdatefcn.m` on the MATLAB path:

```
function txt = myupdatefcn(empt,event_obj)
% Customizes text of data tips

pos = get(event_obj,'Position');
txt = {['Time: ',num2str(pos(1))],...
       ['Amplitude: ',num2str(pos(2))]};
```

To set up and use the update function, type:

```
doc_datacursormode
```

When you place a data tip using this update function, it looks like the one in the following figure.



Alternatives

Use the Data Cursor tool to label *x*, *y*, and *z* values on graphs and surfaces. You can control how data tips display by right-clicking and selecting items from the context menu.

See Also

`brush` | `pan` | `rotate3d` | `zoom`

Tutorials

- “Data Cursor — Displaying Data Values Interactively”

How To

- “Example — Visually Exploring Demographic Statistics”
- “Using Data Cursors with Histograms”

Purpose

Produce short description of input variable

Syntax

```
datatipinfo(var)
```

Description

`datatipinfo(var)` displays a short description of a variable, similar to what is displayed in a datatip in the MATLAB debugger.

Examples

Get datatip information for a 5-by-5 matrix:

```
A = rand(5);

datatipinfo(A)
A: 5x5 double =
    0.4445    0.3567    0.7458    0.0767    0.4400
    0.7962    0.6575    0.3918    0.8289    0.9746
    0.5641    0.9808    0.0265    0.4838    0.6722
    0.9099    0.9653    0.2508    0.4859    0.4054
    0.2857    0.5198    0.7383    0.9301    0.9604
```

Get datatip information for a 50-by-50 matrix. For this larger matrix, `datatipinfo` displays just the size and data type:

```
A = rand(50);

datatipinfo(A)
A: 50x50 double
```

Also for multidimensional matrices, `datatipinfo` displays just the size and data type:

```
A = rand(5);
A(:,:,2) = A(:,:,1);

datatipinfo(A)
A: 5x5x2 double
```

See Also

`inputname` | `narginchk` | `nargin` | `varargin` | `inputParser`

date

Purpose Current date string

Syntax

```
str = date
```

Description `str = date` returns a string containing the date in `dd-mmm-yyyy` format.

See Also `clock` | `datestr` | `datenum` | `now`

Purpose

Convert date and time to serial date number

Syntax

```
DateNumber = datenum(DateString)
DateNumber = datenum(DateString,formatIn)
DateNumber = datenum(DateString,PivotYear)
DateNumber = datenum(DateString,formatIn,PivotYear)

DateNumber = datenum(DateVector)
DateNumber = datenum(Y,M,D)
DateNumber = datenum(Y,M,D,H,MN,S)
```

Description

`DateNumber = datenum(DateString)` converts date strings to serial date numbers. If the date string format is known, use `formatIn`. Syntaxes without `formatIn` are significantly slower than those that include it.

A serial date number represents the whole and fractional number of days from a fixed, preset date (January 0, 0000).

`DateNumber = datenum(DateString,formatIn)` uses `formatIn` to interpret each date string.

`DateNumber = datenum(DateString,PivotYear)` uses `PivotYear` to interpret date strings that specify the year as two characters. If the date string format is known, use `formatIn`. Syntaxes without `formatIn` are significantly slower than those that include it.

`DateNumber = datenum(DateString,formatIn,PivotYear)` uses `formatIn` to interpret each date string, and `PivotYear` to interpret date strings that specify the year as two characters. You can specify `formatIn` and `PivotYear` in either order.

`DateNumber = datenum(DateVector)` converts date vectors to serial date numbers, and returns a column vector of `m` date numbers, where `m` is the total number of date vectors in `DateVector`.

`DateNumber = datenum(Y,M,D)` returns the serial date numbers for corresponding elements of the `Y`, `M`, and `D` (year, month, day) arrays. The arrays must be of the same size (or any can be a scalar). You can also specify the input arguments as a date vector, `[Y,M,D]`.

`DateNumber = datenum(Y,M,D,H,MN,S)` additionally returns the serial date numbers for corresponding elements of the `H`, `MN`, and `S` (hour, minute, and second) arrays. The arrays must be of the same size (or any can be a scalar). You can also specify the input arguments as a date vector, `[Y,M,D,H,MN,S]`.

Input Arguments

DateVector - Date vectors

matrix

Date vectors, specified as an `m`-by-6 or `m`-by-3 matrix containing `m` full or partial date vectors, respectively. A full date vector has six elements, specifying year, month, day, hour, minute, and second, in that order. A partial date vector has three elements, specifying year, month, and day, in that order. Each element of `DateVector` must be a positive double-precision array.

Example: `[2003,10,24,12,45,07]`

Data Types
`double`

DateString - Date strings

string | cell array of strings

Date strings, specified as a single string or a cell array of strings, where each row corresponds to one date string. All of the date strings must have the same format.

Example: `'24 Oct-2003 12:45:07'`

Example: `{'15-Oct-2010';'20-Nov-2012'}`

If the date string format is known, you should also specify `formatIn`. If you do not specify `formatIn`, `DateString` must be in one of the following formats.

Date String Format	Example
'dd-mmm-yyyy HH:MM:SS'	01-Mar-2000 15:45:17
'dd-mmm-yyyy'	01-Mar-2000
'mm/dd/yyyy'	03/01/2000
'mm/dd/yy'	03/01/00
'mm/dd'	03/01
'mmm.dd,yyyy HH:MM:SS'	Mar.01,2000 15:45:17
'mmm.dd,yyyy'	Mar.01,2000
'yyyy-mm-dd HH:MM:SS'	2000-03-01 15:45:17
'yyyy-mm-dd'	2000-03-01
'yyyy/mm/dd'	2000/03/01
'HH:MM:SS'	15:45:17
'HH:MM:SS PM'	3:45:17 PM
'HH:MM'	15:45
'HH:MM PM'	3:45 PM

Certain date string formats might not contain enough information to convert the date string. In those cases, hours, minutes, and seconds default to 0, days default to 1, months default to January, and years default to the current year. MATLAB considers two-character date string years (e.g., `'79'`) to fall within the 100-year range centered around the current year.

When you do not specify `formatIn`, note the following:

- For the formats that specify the month as two digits (`mm`), the month value must not be greater than 12.
- However, for the format `'mm/dd/yy'`, if the first entry in the date string is greater than 12 and the second entry is less than or equal to 12, MATLAB considers the date string to be in `'yy/mm/dd'` format.

formatIn - Format of the input date string

string

Format of the input date string, specified as a string of symbolic identifiers.

Example: `'dddd, mmm dd, yyyy'`

The following table shows symbolic identifiers you can use to construct the `formatIn` string. You can include characters such as a hyphen, space, or colon to separate the fields.

Symbolic Identifier	Description	Example
`yyyy`	Year in full	1990, 2002
`yy`	Year in two digits	90, 02
`QQ`	Quarter year using letter `Q` and one digit	`Q1`
`mmmm`	Month using full name	`March`, `December`
`mmm`	Month using first three letters	`Mar`, `Dec`
`mm`	Month in two digits	03, 12
`m`	Month using capitalized first letter	`M`, `D`
`dddd`	Day using full name	`Monday`, `Tuesday`
`ddd`	Day using first three letters	`Mon`, `Tue`
`dd`	Day in two digits	05, 20
`d`	Day using capitalized first letter	`M`, `T`

Symbolic Identifier	Description	Example
`HH`	Hour in two digits (no leading zeros when symbolic identifier `AM` or `PM` is used)	`05`, `5 AM`
`MM`	Minute in two digits	`12`, `02`
`SS`	Second in two digits	`07`, `59`
`FFF`	Millisecond in three digits	`057`
`AM or PM`	`AM` or `PM` inserted in date string	`3:45:02 PM`

The `formatIn` string must follow these guidelines:

- You cannot specify any field more than once. For example, you cannot use `'yy-mmm-dd-m'` because it has two month identifiers. The one exception to this is that you can combine one instance of `dd` with one instance of any of the other day identifiers. For example, `'dddd mmm dd yyyy'` is a valid input.
- When you use `AM` or `PM`, the `HH` field is also required.
- You only can use `QQ` alone or with a year specifier.

PivotYear - Start year of 100-year date range

current year minus 50 years (default) | integer

Start year of the 100-year date range in which a two-character year resides, specified as an integer. Use a pivot year to interpret date strings that specify the year as two characters.

Example: `2000`

Data Types

`double`

Y,M,D - Year, month, and day arrays

scalar | vector

Year, month, and day arrays, specified as scalars or vectors. The arrays must be of the same size (or any can be a scalar).

You can also specify the input arguments as a date vector, `[Y,M,D]`.

Example: `2003,10,24`

Data Types
`double`

Y,M,D,H,MN,S - Year, month, day, hour, minute and second arrays

scalar | vector

Year, month, day, hour, minute and second arrays, specified as scalars or vectors. The arrays must be of the same size (or any can be a scalar).`datenum` does not accept milliseconds as a separate input, but as a fractional part of the seconds input, `s`.

You can also specify the input arguments as a date vector, `[Y,M,D,H,MN,S]`.

Example: `2003,10,24,12,45,07.451`

Data Types
`double`

Output Arguments

DateNumber - Serial date numbers

scalar | vector

Serial date numbers, returned as a column vector of length `m`, where `m` is the total number of input date vectors or date strings.

Examples

Convert Date String to Date Number

```
DateString = '19-May-2001';
formatIn = 'dd-mmm-yyyy';
datenum(DateString,formatIn)
```

```
ans =
      730990
```

`datenum` returns a date number for the date string with the format `'dd-mmm-yyyy'` .

Convert Multiple Date Strings to Date Numbers

Pass multiple date strings in a cell array. All input date strings must use the same format.

```
DateString = {'09/16/2007';'05/14/1996';'11/29/2010'};
formatIn = 'mm/dd/yyyy';
datenum(DateString,formatIn)
```

```
ans =
    733301
    729159
    734471
```

Convert Date String to Date Number Using Pivot Year

Convert a date string to a serial date number using the default pivot year.

```
n = datenum('12-jun-17','dd-mmm-yy')
```

```
n =
      736858
```

The corresponding date string to this date number is 12-Jun-2017.

Convert the same date string to a serial date number using 1400 as the pivot year.

```
n = datenum('12-jun-17','dd-mmm-yy',1400)
```

```
n =
      517712
```

The corresponding date string to this date number is 12-Jun-1417.

Convert Date Vector to Date Number

```
datenum([2009,4,2,11,7,18])

ans =
   7.3387e+05
```

Convert Year, Month, and Day to Date Number

Convert a date specified by year, month and day values to a serial date number.

```
n = datenum(2001,12,19)

n =
      731204
```

See Also

datestr | datevec

Concepts

- “Represent Dates and Times in MATLAB”
- “Carryover in Date Vectors and Strings”

Purpose

Convert date and time to string format

Syntax

```
DateString = datestr(DateVector)
DateString = datestr(DateNumber)

DateString = datestr( ___ ,formatOut)

DateString = datestr(DateStringIn)
DateString = datestr(DateStringIn,formatOut,PivotYear)

DateString = datestr( ___ ,'local')
```

Description

`DateString = datestr(DateVector)` converts date vectors to date strings. `datestr` returns a column vector of `m` date strings, where `m` is the total number of date vectors in `DateVector`. `datestr` returns date strings in the default date string format `dd-mmm-yyyy HH:MM:SS` (day-month-year hour:minute:second).

`DateString = datestr(DateNumber)` converts serial date numbers to date strings. `datestr` returns a column vector of `m` date strings, where `m` is the total number of date numbers in `DateNumber`.

`DateString = datestr( ___ ,formatOut)` specifies the format of the output date strings using `formatOut`. You can use `formatOut` with any of the input arguments in the above syntaxes.

`DateString = datestr(DateStringIn)` converts `DateStringIn` to date strings in the default date string format `dd-mmm-yyyy HH:MM:SS`. All date strings in `DateStringIn` must have the same format.

`DateString = datestr(DateStringIn,formatOut,PivotYear)` converts `DateStringIn` to `DateString`, in the format specified by `formatOut`, and using optional `PivotYear` to interpret date strings that specify the year as two characters.

DateString = datestr(___ ,'local') returns the date string in the language of the current locale. This is the language you select by means of your computer's operating system. If you leave local out of the argument list, datestr returns the date string in the default language, which is US English. Use local with any of the previous syntaxes. The local argument must be last in the argument sequence.

Input Arguments

DateVector - Date vectors

matrix

Date vectors, specified as an m-by-6 matrix, where m is the number of full (six-element) date vectors. Each element of DateVector must be a positive double-precision number.

Example: [2003,10,24,12,45,07]

Data Types
double

DateNumber - Serial date numbers

scalar | vector | multidimensional array

Serial date numbers, specified as a scalar, vector, or multidimensional array of positive double-precision numbers.

Example: 731878

Data Types
double

formatOut - Format of the date string output

0 (default) | string | integer

Format of the date string output, specified as a string of symbolic identifiers or an integer that corresponds to a predefined format. If you do not specify formatOut, datestr returns a date string in the default date string format dd-mmm-yyyy HH:MM:SS.

The following table shows symbolic identifiers you can use to construct the formatOut string. You can include characters such as a hyphen, space, or colon to separate the fields.

Symbolic Identifier	Description	Example
`yyyy`	Year in full	`1990`, `2002`
`yy`	Year in two digits	`90`, `02`
`QQ`	Quarter year using letter `Q` and one digit	`Q1`
`mmmm`	Month using full name	`March`, `December`
`mmm`	Month using first three letters	`Mar`, `Dec`
`mm`	Month in two digits	`03`, `12`
`m`	Month using capitalized first letter	`M`, `D`
`dddd`	Day using full name	`Monday`, `Tuesday`
`ddd`	Day using first three letters	`Mon`, `Tue`
`dd`	Day in two digits	`05`, `20`
`d`	Day using capitalized first letter	`M`, `T`
`HH`	Hour in two digits (no leading zeros when symbolic identifier `AM` or `PM` is used)	`05`, `5 AM`
`MM`	Minute in two digits	`12`, `02`
`SS`	Second in two digits	`07`, `59`
`FFF`	Millisecond in three digits	`057`
`AM or PM`	`AM` or `PM` inserted in date string	`3:45:02 PM`

The `formatOut` string must follow these guidelines:

- You cannot specify any field more than once. For example, you cannot use 'yy-mmm-dd-m' because it has two month identifiers. The one exception to this is that you can combine one instance of dd with one instance of any of the other day identifiers. For example, 'dddd mmm dd yyyy' is a valid input.
- When you use AM or PM, the HH field is also required.
- You only can use QQ alone or with a year specifier.

The following table lists predefined MATLAB date formats.

Numeric Identifier	Date String Format	Example
0 or -1 (default)	'dd-mmm-yyyy HH:MM:SS'	01-Mar-2000 15:45:17
1	'dd-mmm-yyyy'	01-Mar-2000
2	'mm/dd/yy'	03/01/00
3	'mmm'	Mar
4	'm'	M
5	'mm'	03
6	'mm/dd'	03/01
7	'dd'	01
8	'ddd'	Wed
9	'd'	W
10	'yyyy'	2000
11	'yy'	00
12	'mmmyy'	Mar00
13	'HH:MM:SS'	15:45:17
14	'HH:MM:SS PM'	3:45:17 PM
15	'HH:MM'	15:45

Numeric Identifier	Date String Format	Example
16	'HH:MM PM'	3:45 PM
17	'QQ-YY'	Q1-01
18	'QQ'	Q1
19	'dd/mm'	01/03
20	'dd/mm/yy'	01/03/00
21	'mmm.dd,yyyy HH:MM:SS'	Mar.01,2000 15:45:17
22	'mmm.dd,yyyy'	Mar.01,2000
23	'mm/dd/yyyy'	03/01/2000
24	'dd/mm/yyyy'	01/03/2000
25	'yy/mm/dd'	00/03/01
26	'yyyy/mm/dd'	2000/03/01
27	'QQ-YYYY'	Q1-2001
28	'mmmyyyy'	Mar2000
29	'yyyy-mm-dd' (ISO 8601)	2000-03-01
30	'yyyymmddTHHMMSS' (ISO 8601)	20000301T154517
31	'yyyy-mm-dd HH:MM:SS'	2000-03-01 15:45:17

DateStringIn - Date strings to convert
string | cell array

Date strings to convert, specified as a single string or a cell array of strings, where each row corresponds to one date string.

MATLAB considers two-character date string years (for example, `'79'`) to fall within the 100-year range centered around the current year.

All date strings must have the same date format, and they must be in one of the following MATLAB date formats.

Date String Format	Example
'dd-mmm-yyyy HH:MM:SS'	01-Mar-2000 15:45:17
'dd-mmm-yyyy'	01-Mar-2000
'mm/dd/yyyy'	03/01/2000
'mm/dd/yy'	03/01/00
'mm/dd'	03/01
'mmm.dd,yyyy HH:MM:SS'	Mar.01,2000 15:45:17
'mmm.dd,yyyy'	Mar.01,2000
'yyyy-mm-dd HH:MM:SS'	2000-03-01 15:45:17
'yyyy-mm-dd'	2000-03-01
'yyyy/mm/dd'	2000/03/01
'HH:MM:SS'	15:45:17
'HH:MM:SS PM'	3:45:17 PM
'HH:MM'	15:45
'HH:MM PM'	3:45 PM

Note When converting from one date string format to another, you should first pass the strings to the `datenum` function, so that you can specify the format of the input date strings. This ensures that the format of the input date strings is correctly interpreted. For example, see "Convert Date String from Custom Format" on page 1-1193.

PivotYear - Start year of 100-year date range

current year minus 50 years (default) | integer

Start year of the 100-year date range in which a two-character year resides, specified as an integer. Use a pivot year to interpret date strings that specify the year as two characters.

Example: 2000

Data Types
`double`

Output Arguments

DateString - Date strings

string | two-dimensional character array

Date strings, returned as a character array with `m` rows, where `m` is the total number of input date vectors, serial date numbers, or date strings. The default output date string format is `dd-mmm-yyyy HH:MM:SS` (day-month-year hour:minute:second).

Tips

- To convert a date string not in a predefined MATLAB date format, first convert the date string to a date number, using either `datenum` or `datevec`.

Examples

Convert Date Vector to Date String

```
DateVector = [2009,4,2,11,7,18];

datestr(DateVector)
```

```
ans =
   02-Apr-2009 11:07:18
```

`datestr` returns a date string in the default date string format.

Convert Date and Time to Specific Format

Format the current date in the `mm/dd/yy` format.

You can specify this format using a string of symbolic identifiers.

```
formatOut = 'mm/dd/yy';
datestr(now,formatOut)
```

```
ans =
   09/07/12
```

Alternatively, you can specify this format using a numeric identifier.

```
formatOut = 2;
datestr(now,formatOut)
```

```
ans =
   09/07/12
```

You can reformat the date and time, and also show milliseconds.

```
dt = datestr(now,'mmmm dd, yyyy HH:MM:SS.FFF AM')
```

```
dt =
January 05, 2012  2:44:48.753 PM
```

Convert 12-Hour Time String to 24-Hour Equivalent

Convert the 12-hour time 05:32 p.m. to its 24-hour equivalent.

```
datestr('05:32 PM','HH:MM')
```

```
ans =

17:32
```

Convert the 24-hour time 05:32 to its 12-hour equivalent.

```
datestr('05:32','HH:MM PM')
```

```
ans =

5:32 AM
```

The use of `AM` or `PM` in the `formatOut` string does not influence which characters actually become part of the date string; they only determine

whether or not to include them in the date string. MATLAB selects AM or PM based on the time entered.

Convert Date String from Custom Format

Call datenum inside of datestr to specify the format of the input date string.

```
formatOut = 'dd mmm yyyy';
datestr(datenum('16-04-55','dd-mm-yy',1900),formatOut)

ans =
16 Apr 1955
```

Convert Multiple Date Strings

Convert more than one date string input by passing the multiple date strings in a cell array.

All input date strings must use the same format. For example, the following command passes three dates that all use the mm/dd/yyyy format.

```
datestr(datenum({'09/16/2007';'05/14/1996';'11/29/2010'}, ...
    'mm/dd/yyyy'))

ans =
    16-Sep-2007
    14-May-1996
    29-Nov-2010
```

datestr returns a character array of converted date strings in the default date string format.

Convert Date String with Values Outside Normal Range

Call datenum inside of datestr to return the expected value, because the date below uses a value outside its normal range (month=13).

```
datestr(datenum('13/24/88','mm/dd/yy'))
```

```
ans =
   24-Jan-1989
```

Use a Pivot Year

Change the pivot year to change the year range.

Use a pivot year of 1900.

```
DateStringIn = '4/16/55';
formatOut = 1;
PivotYear = 1900;
datestr(DateStringIn,formatOut,PivotYear)
```

```
ans =
   16-Apr-1955
```

For the same date string, use a pivot year of 2000.

```
PivotYear = 2000;
datestr(DateStringIn,formatOut,PivotYear)
```

```
ans =
   16-Apr-2055
```

Return Date String in Local Language

Convert a date number to a date string in the language of the current locale.

Use the `local` argument in a French locale.

```
DateNumber = 725935;
formatOut = 'mmmm-dd-yyyy';
str = datestr(DateNumber,formatOut,'local')
```

```
str =
Juillet-17-1987
```

You can make the same call without specifying `` `local' ``.

```
str = datestr(DateNumber,formatOut)

str =
July-17-1987
```

In this case, the output defaults to the English language.

See Also

datenum | datevec

Concepts

- "Represent Dates and Times in MATLAB"
- "Troubleshooting: Converting Date Vector Returns Unexpected Output"

datetick

Purpose

Date formatted tick labels

Syntax

```
datetick(tickaxis)
datetick(tickaxis,dateFormat)

datetick( ___ ,'keeplimits')

datetick( ___ ,'keepticks')

datetick(axes_handle, ___ )
```

Description

`datetick(tickaxis)` labels the tick lines of the axis specified by `tickaxis` using dates, replacing the default numeric labels. `datetick` selects a label format based on the minimum and maximum limits of the specified axis. The axis data values should be serial date numbers, as returned by the `datenum` function.

`datetick(tickaxis,dateFormat)` formats the labels according to the string `dateFormat`.

`datetick( ___ ,'keeplimits')` changes the tick labels to date-based labels while preserving the axis limits. Append `'keeplimits'` to any of the previous syntaxes.

`datetick( ___ ,'keepticks')` changes the tick labels to date-based labels while preserving their locations. Append `'keepticks'` to any of the previous syntaxes.

`datetick(axes_handle, ___ )` labels the tick lines of an axis on the axes specified by `axes_handle`. The `axes_handle` argument can precede any of the input argument combinations in the previous syntaxes.

Input Arguments

tickaxis - Axis to label

'x' (default) | 'y' | 'z'

Axis to label with dates, specified as 'x', 'y', or 'z'.

dateFormat - Format of tick line labels

string | integer

Format of the tick line labels, specified as a string of symbolic identifiers or an integer that corresponds to a predefined format.

The following table shows symbolic identifiers you can use to construct the dateFormat string. You can include characters such as a hyphen, space, or colon to separate the fields. For example, to display the day of the month followed by the three-letter abbreviation of the day of the week in parentheses, use dateFormat = 'dd (ddd)'.

Symbolic Identifier	Description	Example
yyyy	Year in full	1990, 2002
yy	Year in two digits	90, 02
QQ	Quarter year using letter Q and one digit	Q1
mmmm	Month using full name	March, December
mmm	Month using first three letters	Mar, Dec
mm	Month in two digits	03, 12
m	Month using capitalized first letter	M, D
dddd	Day using full name	Monday, Tuesday
ddd	Day using first three letters	Mon, Tue
dd	Day in two digits	05, 20

Symbolic Identifier	Description	Example
d	Day using capitalized first letter	M, T
HH	Hour in two digits (no leading zeros when symbolic identifier AM or PM is used)	05, 5 AM
MM	Minute in two digits	12, 02
SS	Second in two digits	07, 59
FFF	Millisecond in three digits	057
AM or PM	AM or PM inserted in date string	3:45:02 PM

The following table lists predefined MATLAB date formats.

Numeric Identifier	Date String Format	Example
0 or -1 (default)	'dd-mmm-yyyy HH:MM:SS'	01-Mar-2000 15:45:17
1	'dd-mmm-yyyy'	01-Mar-2000
2	'mm/dd/yy'	03/01/00
3	'mmm'	Mar
4	'm'	M
5	'mm'	03
6	'mm/dd'	03/01
7	'dd'	01
8	'ddd'	Wed
9	'd'	W

Numeric Identifier	Date String Format	Example
10	'yyyy'	2000
11	'yy'	00
12	'mmmyy'	Mar00
13	'HH:MM:SS'	15:45:17
14	'HH:MM:SS PM'	3:45:17 PM
15	'HH:MM'	15:45
16	'HH:MM PM'	3:45 PM
17	'QQ-YY'	Q1-01
18	'QQ'	Q1
19	'dd/mm'	01/03
20	'dd/mm/yy'	01/03/00
21	'mmm.dd,yyyy HH:MM:SS'	Mar.01,2000 15:45:17
22	'mmm.dd,yyyy'	Mar.01,2000
23	'mm/dd/yyyy'	03/01/2000
24	'dd/mm/yyyy'	01/03/2000
25	'yy/mm/dd'	00/03/01
26	'yyyy/mm/dd'	2000/03/01
27	'QQ-YYYY'	Q1-2001
28	'mmmyyyy'	Mar2000
29	'yyyy-mm-dd' (ISO 8601)	2000-03-01
30	'yyyymmddTHHMMSS' (ISO 8601)	20000301T154517
31	'yyyy-mm-dd HH:MM:SS'	2000-03-01 15:45:17

Examples

Label x-Axis Ticks with 2-digit Years

Graph population data for the 20th Century taken from the 1990 US census and label *x*-axis ticks with 2-digit years.

Create time data by decade.

```
t = (1900:10:1990)';
```

Enter total population counts for the USA.

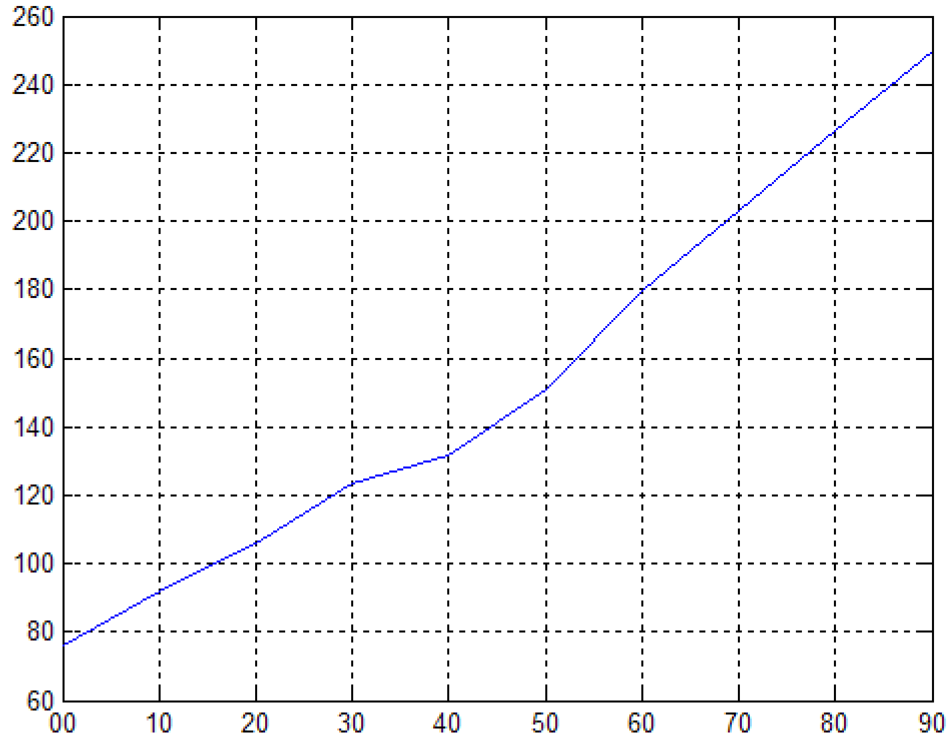
```
p = [75.995 91.972 105.711 123.203 131.669 ...
 150.697 179.323 203.212 226.505 249.633]';
```

Convert years to serial date numbers using the `datenum` function, and then plot the data.

```
figure
plot(datenum(t,1,1),p)
grid on
```

Replace *x*-axis ticks with 2-digit years. The numeric identifier `11` corresponds to the predefined MATLAB date format `'yy'`.

```
dateFormat = 11;
datetick('x',dateFormat)
```



Label x-Axis Ticks with Hours of the Day

Plot traffic count data against date ticks for hours of the day showing AM and PM.

Get traffic count data.

```
load count.dat
```

Create arrays for an arbitrary date, for example, April 18, 1995

```
n = length(count);
year = 1990 * ones(1,n);
month = 4 * ones(1,n);
```

```
day = 18 * ones(1,n);
```

Create arrays for each of 24 hours.

```
hour = 1:n;
minutes = zeros(1,n);
```

Get the serial date numbers for the date arrays.

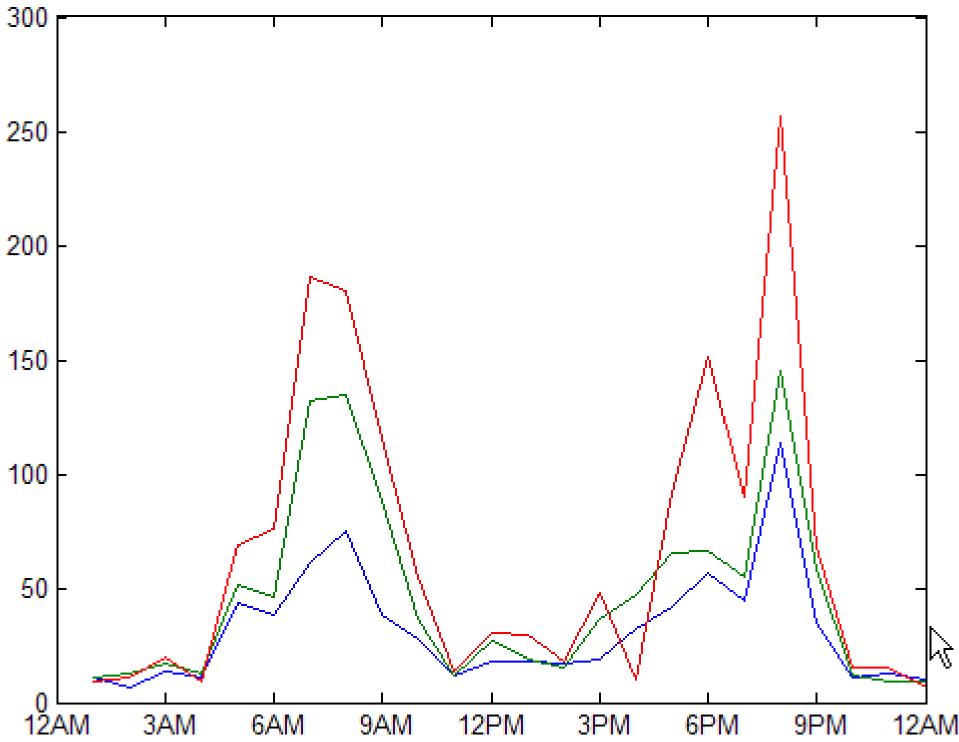
```
xdate = datenum(year,month,day,hour,minutes,minutes);
```

Plot the traffic data against the serial date numbers.

```
figure
plot(xdate,count)
```

Label the tick lines of the graph's *x*-axis with the hours of the day.

```
datetick('x','HHPM')
```



Label x-Axis and Preserve Axis Limits

Select a starting date.

```
startDate = datenum('02-01-1962');
```

Select an ending date.

```
endDate = datenum('11-15-2012');
```

Create a variable, `xdata`, that corresponds to the number of years between the start and end dates.

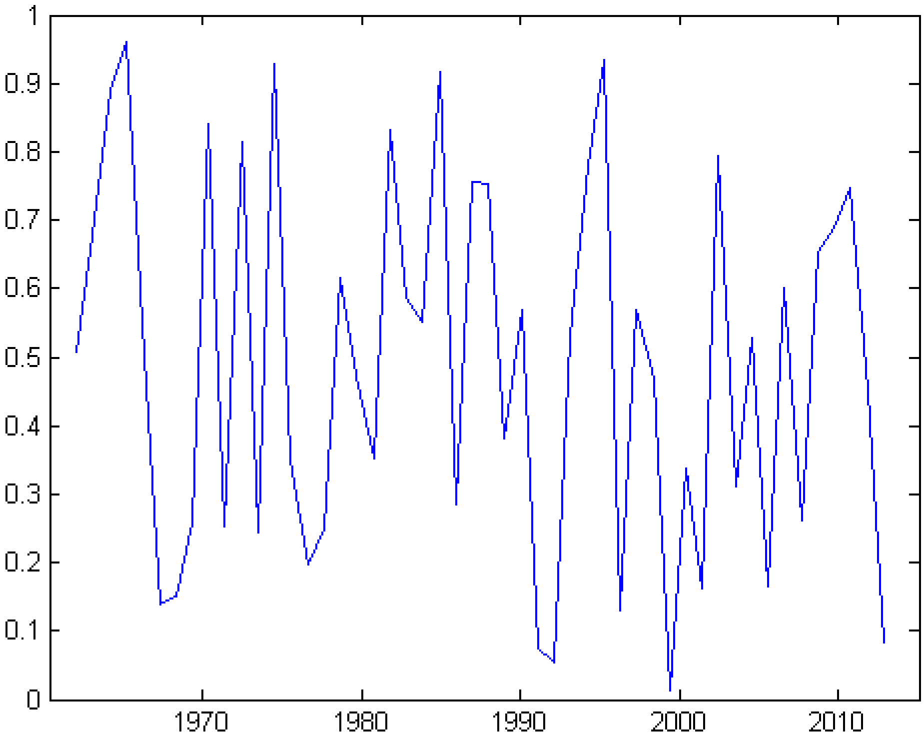
```
xData = linspace(startDate,endDate,50);
```

For this example, plot random data.

```
figure
plot(xData,rand(1,50))
```

Label the x-axis with 4-digit years, preserving the x-axis limits by using the `'keeplimits'` option.

```
datetick('x','yyyy','keeplimits')
```



Add Month Labels to a Plot and Preserve Number of Ticks

Select a starting date.

```
startDate = datenum('01-01-2009');
```

Select an ending date.

```
endDate = datenum('12-31-2009');
```

Create a variable, `xdata`, that corresponds to the number of months between the start and end dates.

```
xData = linspace(startDate,endDate,12);
```

For this example, plot random data.

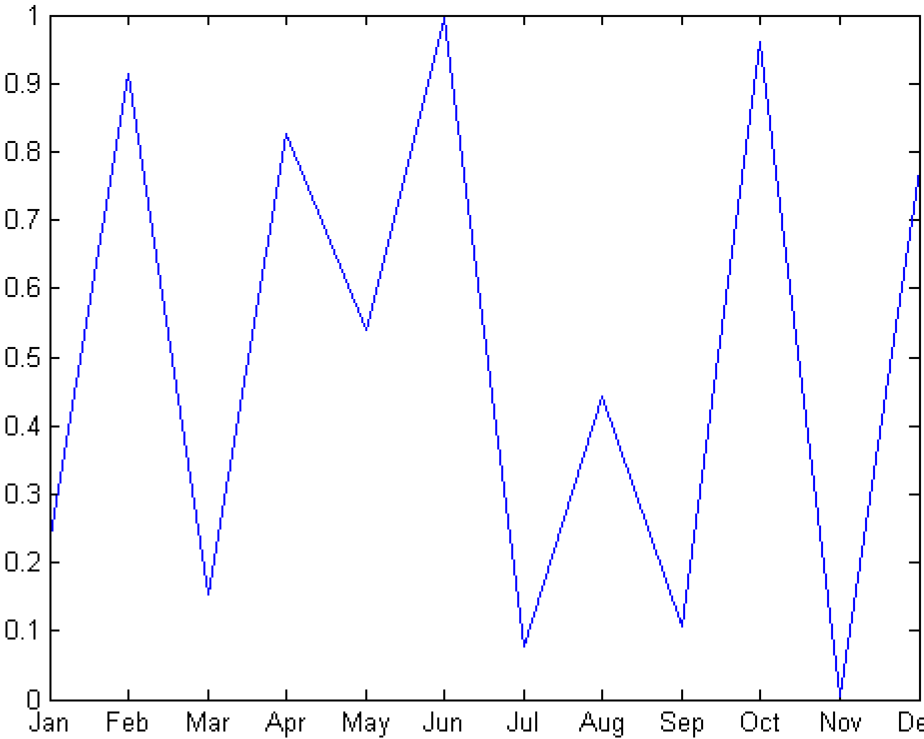
```
figure
plot(xData,rand(1,12))
```

Set the number of `XTicks` to the number of points in `xData`.

```
set(gca,'XTick',xData)
```

Label the *x*-axis with month names, preserving the total number of ticks by using the `'keepticks'` option.

```
datetick('x','mmm','keepticks')
```



Create Multiple Plots Within One Figure and Label Axis with Month Names

Select a starting date and an ending date.

```
startDate = datenum('01-01-2009');
endDate = datenum('12-31-2009');
```

Create a variable, `xdata`, that corresponds to the number of months between the start and end dates.

```
xData = linspace(startDate,endDate,12);
```

For this example, plot random data.

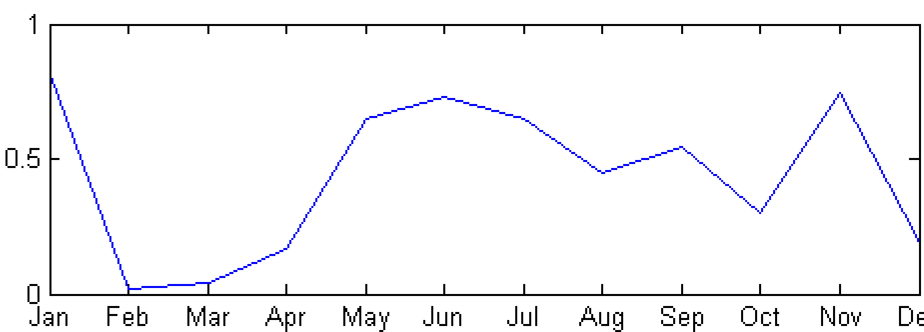
```
figure
s(1)=subplot(2,1,1);
plot(xData,rand(1,12))
s(2) = subplot(2,1,2);
plot(xData,rand(1,12))
```

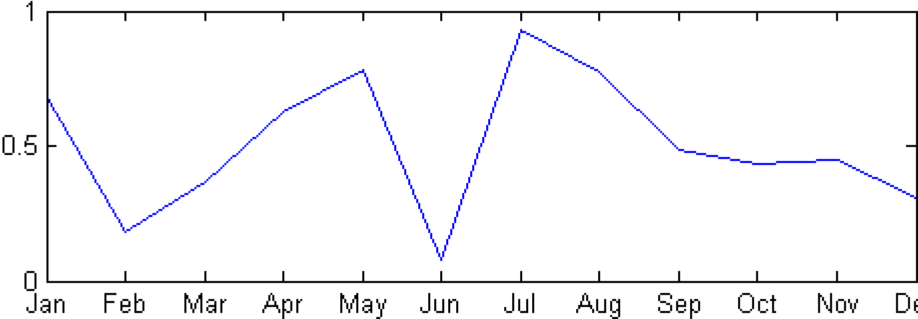
Set the number of `XTicks` to the number of points in `xData`.

```
set(s,'XTick',xData)
```

Label the *x*-axis of each subplot with month names, referring to each subplot using its axes handle. Preserve the total number of ticks by using the `'keepticks'` option.

```
for i = 1:2
 datetick(s(i),'x','mmm','keepticks')
end
```





Algorithms

`datetick` calls the `datestr` function to convert date numbers to date strings.

Tips

- To change the tick spacing and locations, set the appropriate axes property (i.e., `XTick`, `YTick`, or `ZTick`) before calling `datetick`.
- Calling `datetick` sets the `TickMode` of the specified axis to `'manual'`. This means that after zooming, panning or otherwise changing axis limits, you should call `datetick` again to update the ticks and labels.

See Also XTick | YTick | ZTick | datenum | datestr

Purpose

Convert date and time to vector of components

Syntax

```
DateVector = datevec(DateNumber)

DateVector = datevec(DateString)
DateVector = datevec(DateString,formatIn)
DateVector = datevec(DateString,PivotYear)
DateVector = datevec(DateString,formatIn,PivotYear)

[Y, M, D, H, MN, S] = datevec(___)
```

Description

`DateVector = datevec(DateNumber)` converts one or more date numbers to date vectors. `datevec` returns an `m`-by-6 matrix containing `m` date vectors, where `m` is the total number of date numbers in `DateNumber`.

`DateVector = datevec(DateString)` converts date strings to date vectors. If the date string format is known, use `formatIn`. Syntaxes without `formatIn` are significantly slower than those that include it.

`DateVector = datevec(DateString,formatIn)` uses `formatIn` to interpret each date string.

`DateVector = datevec(DateString,PivotYear)` uses `PivotYear` to interpret date strings that specify the year as two characters. If the date string format is known, use `formatIn`. Syntaxes without `formatIn` are significantly slower than those that include it.

`DateVector = datevec(DateString,formatIn,PivotYear)` uses `formatIn` to interpret each date string, and `PivotYear` to interpret date strings that specify the year as two characters. You can specify `formatIn` and `PivotYear` in either order.

[Y, M, D, H, MN, S] = datevec(___) returns the components of the date vector as individual variables Y, M, D, H, MN, and S (year, month, day, hour, minutes and seconds). datevec returns milliseconds as a fractional part of the seconds (S) output.

Input Arguments

DateNumber - Serial date number

scalar | vector | multidimensional array

Serial date number, specified as a scalar, vector, or multidimensional array of positive double-precision numbers.

Example: 731878

Data Types

double

DateString - Date strings

string | cell array of strings

Date strings, specified as a single string or a cell array of strings, where each row corresponds to one date string. All of the date strings must have the same format.

Example: '24 Oct-2003 12:45:07'

Example: {'15-Oct-2010';'20-Nov-2012'}

If the date string format is known, you should also specify formatIn. If you do not specify formatIn, DateString must be in one of the following formats.

Date String Format	Example
'dd-mmm-yyyy HH:MM:SS'	01-Mar-2000 15:45:17
'dd-mmm-yyyy'	01-Mar-2000
'mm/dd/yyyy'	03/01/2000
'mm/dd/yy'	03/01/00
'mm/dd'	03/01

Date String Format	Example
'mmm.dd,yyyy HH:MM:SS'	Mar.01,2000 15:45:17
'mmm.dd,yyyy'	Mar.01,2000
'yyyy-mm-dd HH:MM:SS'	2000-03-01 15:45:17
'yyyy-mm-dd'	2000-03-01
'yyyy/mm/dd'	2000/03/01
'HH:MM:SS'	15:45:17
'HH:MM:SS PM'	3:45:17 PM
'HH:MM'	15:45
'HH:MM PM'	3:45 PM

Certain date string formats might not contain enough information to convert the date string. In those cases, hours, minutes, and seconds default to 0, days default to 1, months default to January, and years default to the current year. MATLAB considers two-character date string years (e.g., `'79'`) to fall within the 100-year range centered around the current year.

When you do not specify `formatIn`, note the following:

- For the formats that specify the month as two digits (`mm`), the month value must not be greater than 12.
- However, for the format `'mm/dd/yy'`, if the first entry in the date string is greater than 12 and the second entry is less than or equal to 12, MATLAB considers the date string to be in `'yy/mm/dd'` format.

formatIn - Format of the input date string

string

Format of the input date string, specified as a string of symbolic identifiers.

The following table shows symbolic identifiers you can use to construct the `formatIn` string. You can include characters such as a hyphen, space, or colon to separate the fields.

Symbolic Identifier	Description	Example
`yyyy`	Year in full	1990, 2002
`yy`	Year in two digits	90, 02
`QQ`	Quarter year using letter `Q` and one digit	Q1
`mmmm`	Month using full name	March, December
`mmm`	Month using first three letters	Mar, Dec
`mm`	Month in two digits	03, 12
`m`	Month using capitalized first letter	M, D
`dddd`	Day using full name	Monday, Tuesday
`ddd`	Day using first three letters	Mon, Tue
`dd`	Day in two digits	05, 20
`d`	Day using capitalized first letter	M, T
`HH`	Hour in two digits (no leading zeros when symbolic identifier `AM` or `PM` is used)	05, 5 AM
`MM`	Minute in two digits	12, 02
`SS`	Second in two digits	07, 59
`FFF`	Millisecond in three digits	057
`AM` or `PM`	`AM` or `PM` inserted in date string	3:45:02 PM

The `formatIn` string must follow these guidelines:

- You cannot specify any field more than once. For example, you cannot use `'yy-mmm-dd-m'` because it has two month identifiers. The one exception to this is that you can combine one instance of `dd` with one instance of any of the other day identifiers. For example, `'dddd mmm dd yyyy'` is a valid input.
- When you use `AM` or `PM`, the `HH` field is also required.
- `datevec` does not accept formats that include `'QQ'`

PivotYear - Start year of 100-year date range
current year minus 50 years (default) | integer

Start year of the 100-year date range in which a two-character year resides, specified as an integer. Use a pivot year to interpret date strings that specify the year as two characters.

Example: 2000

Data Types
`double`

Output Arguments

DateVector - Date vectors
vector | matrix

Date vectors, returned as an `m`-by-6 matrix, where each row corresponds to one date vector, and `m` is the total number of input date numbers or date strings.

[Y, M, D, H, MN, S] - Components of the date vector
scalar

Components of the date vector (year, month, day, hour, minute, and second), returned as individual scalar variables. Milliseconds are a fractional part of the seconds output.

Examples

Convert Date Number to Date Vector

```
format short g
```

```
n = 733779.651;
datevec(n)

ans =
        2009            1            5           15           37
```

Convert Date String to Date Vector

```
DateString = '28.03.2005';
formatIn = 'dd.mm.yyyy';
datevec(DateString,formatIn)

ans =
        2005            3           28            0            0            0
```

datevec returns a date vector for the date string with the format 'dd.mm.yyyy'.

Convert Multiple Date Strings to Date Vectors

Pass multiple date strings in a cell array. All input date strings must use the same format.

```
DateString = {'09/16/2007';'05/14/1996';'11/29/2010'};
formatIn = 'mm/dd/yyyy';
datevec(DateString,formatIn)

ans =
        2007            9           16            0            0            0
        1996            5           14            0            0            0
        2010           11           29            0            0            0
```

Convert Date with Milliseconds to Date Vector

```
datevec('11:21:02.647','HH:MM:SS.FFF')

ans =
   1.0e+03 *
    2.0120    0.0010    0.0010    0.0110    0.0210    0.0026
```

In the output date vector, milliseconds are a fractional part of the seconds field.

Convert Date String to Date Vector Using Pivot Year

Convert a date string to a date vector using the default pivot year.

```
DateString = '12-jun-17';
formatIn = 'dd-mmm-yy';
DateVector = datevec(DateString,formatIn)

DateVector =

        2017           6          12           0           0
```

Convert the same date string to a date vector using 1800 as the pivot year.

```
DateVector = datevec(DateString,formatIn,1800)

DateVector =

        1817           6          12           0           0
```

Assign Elements of Returned Date Vector

Convert a date string to a date vector and return the components of the date vector.

```
[y, m, d, h, mn, s] = datevec('01.02.12','dd.mm.yy')

y =
        2012
m =
     2
d =
     1
h =
     0
mn =
```

```
s =

     0
```

Limitations

- When computing date vectors, MATLAB sets month values less than 1 to 1. Day values, D, less than 1 are set to the last day of the previous month minus |D|. However, if $0 \leq$ `DateNumber` < 1, then `datevec(DateNumber)` returns a date vector of the form `[0 0 0 H MN S]`, where `H`, `MN`, and `S` are hours, minutes, and seconds, respectively.

Tips

- The vectorized calling syntax can offer significant performance improvement for large arrays.

See Also

`datenum | datestr`

Concepts

- "Represent Dates and Times in MATLAB"
- "Carryover in Date Vectors and Strings"

Purpose

Clear breakpoints

Syntax

```
dbclear all
dbclear in file ...
dbclear in file ... -completenames
dbclear if error ...
dbclear if warning ...
dbclear if naninf
dbclear if infnan
```

Description

`dbclear all` removes all breakpoints in all MATLAB code files, as well as breakpoints set for errors, caught errors, caught error identifiers, warnings, warning identifiers, and `naninf`/`infnan`.

`dbclear in file ...` and `dbclear in file ... -completenames` formats are listed here:

Format	Action
`dbclear in file`	Removes all breakpoints in `file`. `file` must be the name of a MATLAB program file, and can include a MATLAB partial path.
`dbclear in file -completenames`	If the command includes the `-completenames` option, then `file` need not be on the path, as long as it is a fully qualified file name. (On Microsoft Windows platforms, this is a file name that begins with `\\` or with a drive letter followed by a colon. On UNIX platforms, this is a file name that begins with `/` or `~`.) `file` can include a `>` to specify the path to a particular local function or to a nested function within a code file.
`dbclear in file at lineno`	Removes the breakpoint set at line number `lineno` in `file`.
`dbclear in file at lineno@`	Removes the breakpoint set in the anonymous function at line number `lineno` in `file`.

Format	Action
`dbclear in file at lineno@n`	Removes the breakpoint set in the nth anonymous function at line number `lineno` in `file`.
`dbclear in file at locfun`	Removes all breakpoints in local function `locfun` in `file`.

`dbclear if error ...` formats are listed here:

Format	Action
`dbclear if error`	Removes the breakpoints set using the `dbstop if error` and `dbstop if error identifier` statements.
`dbclear if error identifier`	Removes the breakpoint set using `dbstop if error identifier` for the specified `identifier`. Running this produces an error if `dbstop if error` or `dbstop if error all` is set.
`dbclear if caught error`	Removes the breakpoints set using the `dbstop if caught error` and `dbstop if caught error identifier` statements.
`dbclear if caught error identifier`	Removes the breakpoints set using the `dbstop if caught error identifier` statement for the specified identifier. Running this produces an error if `dbstop if caught error` or `dbstop if caught error all` is set.

`dbclear if warning ...` formats are listed here:

`dbclear if warning`	Removes the breakpoints set using the `dbstop if warning` and `dbstop if warning identifier` statements.
`dbclear if warning identifier`	Removes the breakpoint set using `dbstop if warning identifier` for the specified `identifier`. Running this produces an error if `dbstop if warning` or `dbstop if warning all` is set.

`dbclear if naninf` removes the breakpoint set by `dbstop if naninf` or `dbstop if infnan`.

`dbclear` **`if infnan`** removes the breakpoint set by `dbstop` **`if infnan`** or `dbstop` **`if naninf`**.

Tips

The **`at`** and **`in`** keywords are optional.

In the syntax, `file` can be a MATLAB program file, or the path to a function within a file. For example

```
dbclear in foo>myfun
```

clears the breakpoint at the `myfun` function in the file `foo.m`.

See Also

`dbcont` | `dbdown` | `dbquit` | `dbstack` | `dbstatus` | `dbstop` | `dbstep` | `dbtype` | `dbup` | `filemarker`

Related Examples

- “Disable and Clear Breakpoints”

dbcont

Purpose

Resume execution

Syntax

```
dbcont
```

Description

`dbcont` resumes execution of a MATLAB code file from a breakpoint. Execution continues until another breakpoint is encountered, a pause condition is met, an error occurs, or MATLAB software returns to the base workspace prompt.

Note If you want to edit a file as a result of debugging, it is best to first quit debug mode and then edit and save changes to the file. If you edit a file while paused in debug mode, you can get unexpected results when you resume execution of the file and the results might not be reliable.

See Also

`dbclear` | `dbdown` | `dbquit` | `dbstack` | `dbstatus` | `dbstop` | `dbstep` | `dbtype` | `dbup`

Related Examples

- "Step Through a File"

Purpose

Reverse workspace shift performed by `dbup`, while in debug mode

Syntax

```
dbdown
```

Description

`dbdown` changes the current workspace context to the workspace of the called MATLAB code file when a breakpoint is encountered. You must have issued the `dbup` function at least once before you issue this function. `dbdown` is the opposite of `dbup`.

Multiple `dbdown` functions change the workspace context to each successively executed MATLAB code file on the stack until the current workspace context is the current breakpoint. It is not necessary, however, to move back to the current breakpoint to continue execution or to step to the next line.

See Also

`dbcont` | `dbclear` | `dbquit` | `dbstack` | `dbstatus` | `dbstop` | `dbstep` | `dbtype` | `dbup`

Related Examples

- "Debugging Process and Features"

dblquad

Purpose

Numerically evaluate double integral over rectangle

Compatibility

`dblquad` will be removed in a future release. Use `integral2` instead.

Syntax

```
q = dblquad(fun,xmin,xmax,ymin,ymax)
q = dblquad(fun,xmin,xmax,ymin,ymax,tol)
q = dblquad(fun,xmin,xmax,ymin,ymax,tol,method)
```

Description

`q = dblquad(fun,xmin,xmax,ymin,ymax)` calls the `quad` function to evaluate the double integral `fun(x,y)` over the rectangle `xmin <= x <= xmax`, `ymin <= y <= ymax`. The input argument, `fun`, is a function handle that accepts a vector `x`, a scalar `y`, and returns a vector of integrand values.

“Parameterizing Functions” explains how to provide additional parameters to the function `fun`, if necessary.

`q = dblquad(fun,xmin,xmax,ymin,ymax,tol)` uses a tolerance `tol` instead of the default, which is `1.0e-6`.

`q = dblquad(fun,xmin,xmax,ymin,ymax,tol,method)` uses the quadrature function specified as `method`, instead of the default `quad`. Valid values for `method` are `@quadl` or the function handle of a user-defined quadrature method that has the same calling sequence as `quad` and `quadl`.

Examples

Pass function handle `@integrnd` to `dblquad`:

```
Q = dblquad(@integrnd,pi,2*pi,0,pi);
```

where the function `integrnd.m` is:

```
function z = integrnd(x, y)
z = y*sin(x)+x*cos(y);
```

Pass anonymous function handle `F` to `dblquad`:

```
F = @(x,y)y*sin(x)+x*cos(y);
Q = dblquad(F,pi,2*pi,0,pi);
```

The `integrnd` function integrates `y*sin(x)+x*cos(y)` over the square `pi <= x <= 2*pi`, `0 <= y <= pi`. Note that the integrand can be evaluated with a vector `x` and a scalar `y`.

Nonsquare regions can be handled by setting the integrand to zero outside of the region. For example, the volume of a hemisphere is:

```
dblquad(@(x,y)sqrt(max(1-(x.^2+y.^2),0)), -1, 1, -1, 1)
```

or

```
dblquad(@(x,y)sqrt(1-(x.^2+y.^2)).*(x.^2+y.^2<=1), -1, 1, -1, 1)
```

See Also

`quad2d` | `quad` | `quadgk` | `quadl` | `triplequad` | `function_handle` | `integral` | `integral2` | `integral3`

How To

- "Anonymous Functions"

dbmex

Purpose

Enable MEX-file debugging (on UNIX platforms)

Syntax

```
dbmex on
dbmex off
dbmex stop
```

Description

`dbmex on` enables MEX-file debugging for UNIX platforms.

To use this option, first start the MATLAB software from a debugger by typing `matlab -D`*`debugger`*, where `debugger` is the name of the debugger program. You must invoke `dbmex on` before calling your MEX-file. If you have already loaded the MEX-file, remove it from memory using the `clear` function.

`dbmex off` disables MEX-file debugging.

`dbmex stop` returns to the debugger prompt.

Tips

For more information about debugging MEX-files, see "Debug C/C++ Language MEX-Files".

See Also

`dbclear` | `dbcont` | `dbdown` | `dbquit` | `dbstack` | `dbstatus` | `dbstep` | `dbstop` | `dbtype` | `dbup`

Purpose

Quit debug mode

Syntax

```
dbquit
dbquit('all')
dbquit all
```

Description

`dbquit` terminates debug mode. The Command Window then displays the standard prompt (>>). The file being processed is *not* completed and no results are returned. All breakpoints remain in effect, unless your file contains (and MATLAB has processed) one or more of the following commands: `clear all`, `clear function`, `clear classes`.

If you debug `file1` and step into `file2`, running `dbquit` terminates debugging for both files. However, if you debug `file3` and also debug `file4`, running `dbquit` terminates debugging for `file4`, but `file3` remains in debug mode until you run `dbquit` again.

`dbquit('all')` or the command form, `dbquit all`, ends debugging for all files at once.

Examples

This example illustrates the use of `dbquit` relative to `dbquit('all')`. Set breakpoints in and run `file1` and `file2`:

```
>> dbstop in file1
>> dbstop in file2
>> file1
K>> file2
K>> dbstack
```

MATLAB software returns

```
K>> dbstack
  In file1 at 11
  In file2 at 22
```

If you use the `dbquit` syntax

```
K>> dbquit
```

MATLAB ends debugging for `file2` but `file1` is still in debug mode as shown here

```
K>> dbstack
    in file1 at 11
```

Run `dbquit` again to exit debug mode for `file1`.

Alternatively, `dbquit('all')` ends debugging for both files at once:

```
K>> dbstack
  In file1 at 11
  In file2 at 22
dbquit('all')
dbstack
```

returns no result.

See Also

`clear` | `dbcont` | `dbclear` | `dbdown` | `dbstack` | `dbstatus` | `dbstop` | `dbstep` | `dbtype` | `dbup`

Related Examples

- "Correct Problems and End Debugging"

Purpose

Function call stack

Syntax

```
dbstack
dbstack(n)
dbstack('-completenames')
[ST,I] = dbstack(...)
```

Description

`dbstack` displays the line numbers and file names of the function calls that led to the current breakpoint, listed in the order in which they were executed. The display lists the line number of the most recently executed function call (at which the current breakpoint occurred) first, followed by its calling function, which is followed by its calling function, and so on. This continues until the topmost MATLAB function is reached. Each line number is a hyperlink you can click to go directly to that line in the Editor. The notation `functionname>localfunctionname` is used to describe the location of the local function.

`dbstack(n)` omits the first `n` frames from the display. This is useful when issuing a `dbstack` from within an error handler, for example.

`dbstack('`**`-completenames`**`')` outputs the “complete name“ (the absolute file name and the entire sequence of functions that nests the function in the stack frame) of each function in the stack.

Either none, one, or both `n` and `'`**`-completenames`**`'` can appear. If both appear, the order is irrelevant.

`[ST,I] = dbstack(...)` returns the stack trace information in an `m`-by-`1` structure, `ST`, with the fields:

`file`	The file in which the function appears. This field is the empty string if there is no file.
`name`	Function name within the file.
`line`	Function line number.

The current workspace index is returned in `I`.

If you step past the end of a file, dbstack returns a negative line number value to identify that special case. For example, if the last line to be executed is line 15, then the dbstack line number is 15 before you execute that line and -15 afterwards.

Tips

In addition to using dbstack while debugging, you can also use dbstack within a MATLAB code file outside the context of debugging. In this case, to get and analyze information about the current file stack. For example, to get the name of the calling file, use dbstack with an output argument within the file being called. For example:

```
st=dbstack;
```

Examples

This example shows the information returned when you issue dbstack while debugging a MATLAB code file:

```
dbstack

In /usr/local/matlab/toolbox/matlab/cond.m at line 13
In test1.m at line 2
In test.m at line 3
```

This example shows the information returned when you issue dbstack while debugging lengthofline.m to get the complete name of the file, the function name, and line number in which the function appears:

```
[ST,I] = dbstack('-completenames')
ST =
    file: 'I:\MATLABFiles\mymfiles\lengthofline.m'
    name: 'lengthofline'
    line: 28
I =
     1
```

See Also

dbcont | dbclear | dbdown | dbquit | dbstatus | dbstop | dbstep | dbtype | dbup | evalin | mfilename | whos

Related Examples

- “Examine Values”

dbstatus

Purpose

List all breakpoints

Syntax

```
dbstatus
dbstatus file
dbstatus('-completenames')
s = dbstatus(...)
```

Description

`dbstatus` lists all the breakpoints in effect including errors, caught errors, warnings, and `naninfs`.

`dbstatus file` displays a list of the line numbers for which breakpoints are set in the specified file, where `file` is a MATLAB code file function name or a MATLAB relative partial path. Each line number is a hyperlink you can click to go directly to that line in the Editor.

`dbstatus('-completenames')` displays, for each breakpoint, the absolute file name and the sequence of functions that nest the function containing the breakpoint.

`s = dbstatus(...)` returns breakpoint information in an `m`-by-`1` structure with the fields listed in the following table. Use this syntax to save breakpoint status and restore it at a later time using `dbstop(s)`—see `dbstop` for an example.

`name`	Function name.
`file`	Full path for file containing breakpoints.
`line`	Vector of breakpoint line numbers.
`anonymous`	Vector of integers representing the anonymous functions in the `line` field. For example, `2` means the second anonymous function in that line. A value of `0` means the breakpoint is at the start of the line, not in an anonymous function.
`expression`	Cell vector of breakpoint conditional expression strings corresponding to lines in the `line` field.

`cond`	Condition string (`'error'`, `'caught error'`, `'warning'`, or `'naninf'`).
`identifier`	When `cond` is `'error'`, `'caught error'`, or `'warning'`, a cell vector of MATLAB message identifier strings for which the particular `cond` state is set.

Use `dbstatus class/function`, `dbstatus private/function`, or `dbstatus class/private/function` to determine the status for methods, private functions, or private methods (for a class named `class`).

In all forms you can further qualify the function name with a local function name, as in `dbstatus function>localfunction`.

Tips

In the syntax, `file` can be a file, or the path to a function within a file. For example

```
Breakpoint for foo>myfun is on line 9
```

means there is a breakpoint at the `myfun` local function, which is line 9 in the file `foo.m`.

See Also

`dbclear` | `dbcont` | `dbdown` | `dbquit` | `dbstack` | `dbstop` | `dbstep` | `dbtype` | `dbup` | `error` | `warning`

dbstep

Purpose

Execute one or more lines from current breakpoint

Syntax

```
dbstep
dbstep nlines
dbstep in
dbstep out
```

Description

This function allows you to debug a MATLAB code file by following its execution from the current breakpoint. At a breakpoint, the `dbstep` function steps through execution of the current file one line at a time or at the rate specified by `nlines`.

`dbstep` executes the next executable line of the current file. `dbstep` steps over the current line, skipping any breakpoints set in functions called by that line.

`dbstep nlines` executes the specified number of executable lines.

`dbstep in` steps to the next executable line. If that line contains a call to another MATLAB code file function, execution will step to the first executable line of the called function. If there is no call to a MATLAB code file on that line, `dbstep in` is the same as `dbstep`.

`dbstep out` runs the rest of the function and stops just after leaving the function.

For all forms, MATLAB software also stops execution at any breakpoint it encounters.

Note If you want to edit a file as a result of debugging, it is best to first quit debug mode and then edit and save changes to the file. If you edit a file while paused in debug mode, you can get unexpected results when you resume execution of the file and the results might not be reliable.

See Also

`dbclear` | `dbcont` | `dbdown` | `dbquit` | `dbstack` | `dbstatus` | `dbstop` | `dbtype` | `dbup`

Related Examples

- “Step Through a File”

dbstop

Purpose

Set breakpoints for debugging

Syntax

```
dbstop in file
dbstop in file at location
dbstop in file if expression
dbstop in file at location if expression
dbstop if condition
dbstop(s)
```

Description

`dbstop in file` sets a breakpoint at the first executable line in `file`. When you run `file`, MATLAB enters debug mode and pauses execution at the first executable line.

`dbstop in file at location` sets a breakpoint at the specified location. MATLAB execution pauses immediately before that location, unless the location is an anonymous function. If the location is an anonymous function, then execution pauses just after the breakpoint.

`dbstop in file if expression` sets a breakpoint at the line containing the `expression`. Execution pauses only if that expression evaluates to `1` (true).

`dbstop in file at location if expression` sets a breakpoint at the specified location. Execution pauses at or just before that location only if the `expression` evaluates to true.

`dbstop if condition` pauses execution at the line that meets the specified `condition`.

`dbstop(s)` restores breakpoints you previously saved to `s`. The files containing the saved breakpoints must be on the search path or in the current folder. MATLAB assigns breakpoints by line number; therefore, the lines in the file must be the same as when you saved the breakpoints, or the results are unpredictable.

Tips

- Use `dbcont` or `dbstep` to resume execution after a breakpoint pauses execution. Use `dbquit` to exit debug mode.

- If you debug a file that MATLAB uses when running and debugging files, and that file is not a MATLAB code file, then some debugging features do not operate as expected. For instance, typing `help functionname` at the debug (`K>>`) prompt will not return help.
- MATLAB can become unresponsive when it stops at a breakpoint while displaying a modal dialog box or figure created by your program. Use **Ctrl+C** to exit debug mode and return to the MATLAB prompt (`>>`).
- If MATLAB pauses and displays a hyperlinked line number in the Command Window, click the hyperlink, The file opens in the Editor at the line where MATLAB paused execution. For an example of such a link, see the image in "Stop at File That Is Not a MATLAB Code File" on page 1-1242
- You can set breakpoints only at executable lines in saved files that are in the current folder or in folders on the search path.
- `dbstop` **`if warning`** has no effect when you disable warnings using `warning off all`. Similarly, `dbstop` **`if warning`** `identifier` has no effect when you disable warnings for the specified message identifier. See `warning` for more information about `off`, `all`, and `warning off identifier`

Input Arguments

file

File specification, typically for a MATLAB function, specified as a string. The file specification can include a partial path, but must be in a folder on the search path, or in the current folder.

If the command includes the `-completenames` option, then the file need not be on the search path, as long as the file specification is a "fully qualified name" on page 1-1239. In addition, `filespec` can include a filemarker (`>`) to specify the path to a particular local function or to a nested function within the file.

If `file` is not a MATLAB code file (for instance, it is a built-in or MDL-file), then MATLAB issues a warning. MATLAB cannot stop *in* the file, so it pauses before executing the file.

location

Location in `file` where you want to set a breakpoint, specified as one of the following:

- lineno

  Line number in `file` specified as a string. The default is `1`.

- lineno@n

  `n`th anonymous function on line number, `lineno`, specified as a string. The default value of `n` is `1`.

- subfun

  Name of a local function in `file`, specified as a string.

expression

Code that evaluates (as if by `eval`) to a scalar logical value of `1` or `0`, (true or false, respectively).

condition

Condition that causes execution to pause when that condition evaluates to true. Specify `condition` as one of the following:

- `error`— Run-time error that occurs outside a `try/catch` block. You cannot resume execution after an uncaught run-time error.

  If you want execution to pause only if a specific error occurs, specify the message id. For example:

  - `dbstop if error` stops execution at the first run-time error that occurs outside a `try/catch` block.
  - `dbstop if error MATLAB:ls:InputsMustBeStrings` pauses execution at the first run-time error outside a `try/catch` block that has a message ID of `MATLAB:ls:InputsMustBeStrings`.

- `caught error` — Run-time error that occurs within the `try` portion of a `try/catch` block. If you want execution to stop only if a specific

error occurs, specify the message id. See the `error` condition for an example of specifying a message id.

- `warning`— Run-time warning occurs. If you want execution to pause only if a specific warning occurs, specify the message id. See the `error` condition for an example of specifying a message id.

  This condition has no effect if you disabled warnings with the `warning off all` command or if you disabled warnings for the specified `id`. For more information about disabling warnings, see `warning`.

- `naninf` and `infnan`

  The code returns an infinite value (`Inf`) or a value that is not a number (`NaN`) as a result of an operator, function call, or scalar assignment. The `naninf` and `infnan` conditions have identical effects.

s

Breakpoints previously saved to a structure using `s=dbstatus`.

Definitions

fully qualified name

- On Microsoft Windows platforms, a file name that begins with two back slashes (`\\`) or with a drive letter followed by a colon (`:`).
- On UNIX platforms, a file name that begins with a slash (`/`) or a tilde (`~`).

Examples

Stop at First Executable Line

This example shows how to set a breakpoint and pause execution at the first executable line in `buggy.m`.

1. Create a file, `buggy.m`, which contains these statements.

   ```
   function z = buggy(x)
   n = length(x);
   z = (1:n)./x;
   ```

2. Issue the `dbstop` command.

```
dbstop in buggy
```

3. Run buggy.

```
buggy(2:5)
```

MATLAB displays the line where it pauses and enters debug mode.

```
2   n = length(x);
K>>
```

4. Advance to the next line in the file.

```
dbstep
```

5. Examine the value of `n` in the Variable Editor.

```
openvar('n')
```

6. Quit debug mode.

```
dbquit
```

Stop if Error

This example shows how to set a breakpoint and pause execution if a run-time error occurs.

1. Create a file, `buggy.m`, which contains these statements.

```
function z = buggy(x)
n = length(x);
z = (1:n)./x;
```

2. Issue these statements in the Command Window:

```
dbstop if error
buggy(magic(3))
```

Because buggy.m works on vectors only, the input in step 2 results in a run-time error. MATLAB goes into debug mode, paused at line 3 in buggy.m.

```
Error using  ./
Matrix dimensions must agree.

Error in buggy at 3
z = (1:n)./x;
3   z = (1:n)./x;
```

3 Quit debug mode.

```
dbquit
```

Stop if InfNaN

This example shows how to set a breakpoint and pause execution if the code returns a NaN value.

1 Create a file, buggy.m, containing these statements.

```
function z = buggy(x)
n = length(x);
z = (1:n)./x;
```

2 Issue these commands.

```
dbstop if naninf
buggy(0:2)
```

If any of the elements of the input, x, is zero, a division by zero occurs in buggy.m. Therefore, MATLAB returns the following message.

```
NaN/Inf breakpoint hit for buggy on line 3.
```

MATLAB is in debug mode, paused at line 3.

Stop at Function in File

This example shows how to set a breakpoint in the `collatzall` program file at the first executable line in the `collatzplot_new` function.

1. Copy `collatzall.m` from the MATLAB examples folder to your current folder.

   ```
   copyfile(fullfile(matlabroot,'help','techdoc','matlab_env',...
   'examples','collatzall.m'),'.','f')
   ```

2. Open `collatzall.m` in the Editor.

   ```
   open collatzall.m
   ```

3. Set the breakpoint.

   ```
    dbstop in collatzall>collatzplot_new
   ```

Stop at File That Is Not a MATLAB Code File

This example shows how to set a breakpoint at the built-in function `clear` when you run `myfile.m`.

1. Create a file, `myfile.m`, in the Editor, containing these statements.

   ```
   function myfile(x)
   n = length(x);
   if n < 10
       disp('length is less than 10')
   else
       disp('length is 10 or greater')
   end
   clear
   disp('Value of n is cleared')
   ```

2. Issue these commands to set the breakpoint.

   ```
   x = [1 2 3 4 5 6 7 8 9 10]
   ```

```
dbstop in clear; myfile(x)
```

MATLAB issues a warning, and pauses before the call to the clear function. The warning appears as shown in this image.

```
Warning: MATLAB debugger can only stop in MATLAB code files, a
"m_interpreter>clear" is not a MATLAB code file.
         Instead, the debugger will stop at the point right be
         "m_interpreter>clear" is called.
length is 10 or greater
8    clear
K>>
```

Restore Saved Breakpoints

This example shows how to save, and then restore, saved breakpoints.

1 Copy `collatzall.m` from the MATLAB examples folder to your current folder.

```
copyfile(fullfile(matlabroot,'help','techdoc','matlab_env','examples
'collatzall.m'),'.','f')
```

2 Open `collatzall.m` in the Editor.

```
open collatzall.m
```

3 Set breakpoints from the Command Window.

```
dbstop at 12 in collatzall
dbstop if error
```

4 Run `dbstatus`.

```
dbstatus
```

MATLAB describes the breakpoints you set.

```
Breakpoint for collatzall>collatzplot_new is on line 12.
Stop if error.
```

5. Save the breakpoints to the structure, `s`, and then save `s` to the MAT-file, `myfilebrkpnts`.

   ```
   s=dbstatus('-completenames');
   save myfilebrkpnts s
   ```

   Using `s=dbstatus('-completenames')` saves absolute paths and the breakpoint function nesting sequence.

6. Clear all breakpoints.

   ```
   dbclear collatzall
   ```

7. Restore the breakpoints by loading the MAT-file.

   ```
   load myfilebrkpnts
   dbstop(s)
   ```

   The file (or files) containing the breakpoints must be on the search path or in the current folder.

8. Verify the breakpoints.

   ```
   dbstatus collatzall
   ```

See Also

`dbclear` | `dbcont` | `dbdown` | `dbquit` | `dbstack` | `dbstatus` | `dbstep` | `dbtype` | `dbup`

Related Examples

- “Set Breakpoints”

Purpose

List text file with line numbers

Syntax

```
dbtype filename
dbtype filename start:end
```

Description

The `dbtype` command is used to list a text file with line numbers, which is helpful when setting breakpoints in a MATLAB code file with `dbstop`.

`dbtype filename` displays the contents of the specified text file, with the line number preceding each line. `filename` must be the full path name of a file, or a MATLAB relative partial path.

`dbtype filename start:end` displays the portion of the file specified by a range of line numbers from `start` to `end`.

You cannot use `dbtype` for built-in functions.

Examples

To see only the input and output arguments for a function, that is, the first line of the file, use the syntax

```
dbtype filename 1
```

For example,

```
dbtype addpath 1
```

returns

```
1     function oldpath = addpath(varargin)
```

See Also

`dbclear` | `dbcont` | `dbdown` | `dbquit` | `dbstack` | `dbstatus` | `dbstep` | `dbstop` | `dbup`

dbup

Purpose

Shift current workspace to workspace of caller, while in debug mode

Syntax

```
dbup
```

Description

This function allows you to examine the calling MATLAB code file to determine what caused the arguments to be passed to the called function.

`dbup` changes the current workspace context, while the user is in the debug mode, to the workspace of the calling file.

Multiple `dbup` functions change the workspace context to each previous calling file on the stack until the base workspace context is reached. (It is not necessary, however, to move back to the current breakpoint to continue execution or to step to the next line.)

Tips

If your receive an error message such as the following, it means that the parent workspace is under construction so that the value of `x` is unavailable:

```
??? Reference to a called function result under construction x
```

See Also

`dbclear` | `dbcont` | `dbdown` | `dbquit` | `dbstack` | `dbstatus` | `dbstep` | `dbstop` | `dbtype`

Concepts

- "Problems Viewing Variable Values from the Parent Workspace"

Purpose

Solve delay differential equations (DDEs) with constant delays

Syntax

```
sol = dde23(ddefun,lags,history,tspan)
sol = dde23(ddefun,lags,history,tspan,options)
```

Arguments

`ddefun` — Function handle that evaluates the right side of the differential equations $y'(t) = f\left(t, y(t), y(t-\tau_1), \ldots, y(t-\tau_k)\right)$. The function must have the form

```
dydt = ddefun(t,y,Z)
```

where `t` corresponds to the current t, `y` is a column vector that approximates $y(t)$, and `Z(:,j)` approximates $y(t - \tau_j)$ for delay τ_j = `lags(j)`. The output is a column vector corresponding to $f\left(t, y(t), y(t-\tau_1), \ldots, y(t-\tau_k)\right)$.

`lags` — Vector of constant, positive delays $\tau_1, \ldots, \tau_k$.

`history` — Specify `history` in one of three ways:

- A function of t such that `y = history(t)` returns the solution $y(t)$ for $t \le t_0$ as a column vector
- A constant column vector, if $y(t)$ is constant
- The solution `sol` from a previous integration, if this call continues that integration

`tspan`	Interval of integration from `t0=tspan(1)` to `tf=tspan(end)` with `t0 < tf`.
`options`	Optional integration argument. A structure you create using the `ddeset` function. See `ddeset` for details.

Description

`sol = dde23(ddefun,lags,history,tspan)` integrates the system of DDEs

$$y'(t) = f\left(t, y(t), y(t-\tau_1), \ldots, y(t-\tau_k)\right)$$

on the interval $[t_0, t_f]$, where τ_1, ..., τ_k are constant, positive delays and t_0, t_f. The input argument, `ddefun`, is a function handle.

"Parameterizing Functions" explains how to provide additional parameters to the function `ddefun`, if necessary.

`dde23` returns the solution as a structure `sol`. Use the auxiliary function `deval` and the output `sol` to evaluate the solution at specific points `tint` in the interval `tspan = [t0,tf]`.

```
yint = deval(sol,tint)
```

The structure `sol` returned by `dde23` has the following fields.

`sol.x`	Mesh selected by `dde23`
`sol.y`	Approximation to $y(x)$ at the mesh points in `sol.x`.
`sol.yp`	Approximation to $y'(x)$ at the mesh points in `sol.x`
`sol.solver`	Solver name, `'dde23'`

`sol = dde23(ddefun,lags,history,tspan,options)` solves as above with default integration properties replaced by values in `options`, an argument created with `ddeset`. See `ddeset` and "Types of DDEs" in the MATLAB documentation for details.

Commonly used options are scalar relative error tolerance `'RelTol'` (`1e-3` by default) and vector of absolute error tolerances `'AbsTol'` (all components are `1e-6` by default).

Use the `'Jumps'` option to solve problems with discontinuities in the history or solution. Set this option to a vector that contains the locations of discontinuities in the solution prior to `t0` (the history) or in coefficients of the equations at known values of t after `t0`.

Use the `'Events'` option to specify a function that `dde23` calls to find where functions $g\left(t, y(t), y(t-\tau_1), \ldots, y(t-\tau_k)\right)$ vanish. This function must be of the form

```
[value,isterminal,direction] = events(t,y,Z)
```

and contain an event function for each event to be tested. For the `k`th event function in `events`:

- `value(k)` is the value of the `k`th event function.
- `isterminal(k) = 1` if you want the integration to terminate at a zero of this event function and `0` otherwise.
- `direction(k) = 0` if you want `dde23` to compute all zeros of this event function, `+1` if only zeros where the event function increases, and `-1` if only zeros where the event function decreases.

If you specify the `'Events'` option and events are detected, the output structure `sol` also includes fields:

`sol.xe`	Row vector of locations of all events, i.e., times when an event function vanished
`sol.ye`	Matrix whose columns are the solution values corresponding to times in `sol.xe`
`sol.ie`	Vector containing indices that specify which event occurred at the corresponding time in `sol.xe`

Examples

This example solves a DDE on the interval [0, 5] with lags 1 and 0.2. The function `ddex1de` computes the delay differential equations, and `ddex1hist` computes the history for `t <= 0`.

Note The file, `ddex1.m`, contains the complete code for this example. To see the code in an editor, type `edit ddex1` at the command line. To run it, type `ddex1` at the command line.

```
sol = dde23(@ddex1de,[1, 0.2],@ddex1hist,[0, 5]);
```

This code evaluates the solution at 100 equally spaced points in the interval `[0,5]`, then plots the result.

```
tint = linspace(0,5);
yint = deval(sol,tint);
plot(tint,yint);
```

`ddex1` shows how you can code this problem using local functions. For more examples see `ddex2`.

Algorithms

`dde23` tracks discontinuities and integrates with the explicit Runge-Kutta (2,3) pair and interpolant of `ode23`. It uses iteration to take steps longer than the lags.

References

[1] Shampine, L.F. and S. Thompson, "Solving DDEs in MATLAB," *Applied Numerical Mathematics*, Vol. 37, 2001, pp. 441-458.

[2] Kierzenka, J., L.F. Shampine, and S. Thompson, "Solving Delay Differential Equations with DDE23," available at `www.mathworks.com/dde_tutorial`.

See Also

`ddesd` | `ddensd` | `ddeget` | `ddeset` | `deval` | `function_handle`

Purpose

Extract properties from delay differential equations options structure

Syntax

```
val = ddeget(options,'name')
val = ddeget(options,'name',default)
```

Description

`val = ddeget(options,'name')` extracts the value of the named property from the structure `options`, returning an empty matrix if the property value is not specified in `options`. It is sufficient to type only the leading characters that uniquely identify the property. Case is ignored for property names. `[]` is a valid `options` argument.

`val = ddeget(options,'name',default)` extracts the named property as above, but returns `val = default` if the named property is not specified in `options`. For example,

```
val = ddeget(opts,'RelTol',1e-4);
```

returns `val = 1e-4` if the `RelTol` is not specified in `opts`.

See Also

`dde23` | `ddesd` | `ddensd` | `ddeset`

Purpose

Solve delay differential equations (DDEs) of neutral type

Syntax

```
sol = ddensd(ddefun,dely,delyp,history,tspan)
sol = ddensd(ddefun,dely,delyp,history,tspan,options)
```

Description

`sol = ddensd(ddefun,dely,delyp,history,tspan)` integrates a system of delay differential equations of neutral type, that has the form

$$y'(t) = f(t, y(t), y(dy_1),..., y(dy_p), y'(dyp_1),..., y'(dyp_q)) \quad \textbf{(1-1)}$$

where

- t is the independent variable representing time.
- dy_i is any of p solution delays.
- dyp_j is any of q derivative delays.

`sol = ddensd(ddefun,dely,delyp,history,tspan,options)` replaces default integration parameters with those specified in `options`, a structure created with the `ddeset` function.

Input Arguments

ddefun - Derivative function

function handle

Derivative function, specified as a function handle whose syntax is `yp = ddefun(t,y,ydel,ypdel)`. The arguments for `ddefun` are described in the table below.

ddefun Argument	Description
`t`	A scalar value representing the current value of time, t.
`y`	A vector that represents $y(t)$ in Equation 1-1. The size of this vector is `n`-by-`1`, where `n` is the number of equations in the system you want to solve.

ddefun Argument	Description
`ydel`	A matrix whose columns, `ydel(:,i)`, represent $y(dy_i)$. The size of this matrix is `n`-by-`p`, where `n` is the number of equations in the system you want to solve, and `p` is the number of $y(dy)$ terms in Equation 1-1.
`ypdel`	A matrix whose columns, `ypdel(:,j)` represent $y'(dyp_j)$. The size of this matrix is `n`-by-`q`, where `n` is the number of equations in the system you want to solve, and `q` is the number of $y'(dyp)$ terms in Equation 1-1.
`yp`	The result returned by `ddefun`. It is an `n`-by-`1` vector whose elements represent the right side of Equation 1-1.

dely - Solution delays
function handle | vector

Solution delays, specified as a function handle, which returns $dy_1,..., dy_p$ in Equation 1-1. Alternatively, you can pass constant delays in the form of a vector.

If you specify `dely` as a function handle, the syntax must be `dy = dely(t,y)`. The arguments for this function are described in the table below.

dely Argument	Description
`t`	A scalar value representing the current value of time, t.
`y`	A vector that represents $y(t)$ in Equation 1-1. The size of this vector is `n`-by-`1`, where `n` is the number of equations in the system you want to solve.
`dy`	A vector returned by the `dely` function whose values are the solution delays, dy_i , in Equation 1-1. The size of this vector is `p`-by-`1`, where `p` is the number of solution delays in the equation. Each element must be less than or equal to t.

If you want to specify constant solution delays having the form $dy_i = t - \tau_i$, then `dely` must be a vector, where `dely(i)` = τ_i. Each value in this vector must be greater than or equal to zero.

If dy is not present in the problem, set `dely` to `[]`.

Data Types

`function_handle | single | double`

delyp - Derivative delays

function handle | vector

Derivative delays, specified as a function handle, which returns dyp_1,..., dyp_q in Equation 1-1. Alternatively, you can pass constant delays in the form of a vector.

If `delyp` is a function handle, its syntax must be `dyp = delyp(t,y)`. The arguments for this function are described in the table below.

delyp Argument	Description
t	A scalar value representing the current value of time, t.
y	A vector that represents $y(t)$ in Equation 1-1. The size of this vector is n-by-1, where n is the number of equations in the system you want to solve.
dyp	A vector returned by the delyp function whose values are the derivative delays, dyp_j, in Equation 1-1. The size of this vector must be q-by-1, where q is the number of solution delays, dyp_j, in the equation. Each element of dyp must be less than t. There is one exception to this restriction: if you are solving an initial value DDE, the value of dyp can equal t at $t = t_0$. For more information, see "Initial Value Neutral Delay Differential Equations" on page 1-1257.

If you want specify constant derivative delays having the form $dyp_j = t - \tau_j$, then delyp must be a vector, where delyp(j) = τ_j. Each value in this vector must be greater than zero. An exception to this restriction occurs when you solve initial value problems for DDEs of neutral type. In such cases, a value in delyp can equal zero at $t = t_0$. See "Initial Value Neutral Delay Differential Equations" on page 1-1257 for more information.

If dyp is not present in the problem, set delyp to [].

Data Types

function_handle | single | double

history - Solution history

function handle | column vector | structure (sol, from previous integration) | 1-by-2 cell array

Solution history, specified as a function handle, column vector, `sol` structure (from a previous integration), or a cell array. This is the solution at $t \leq t_0$.

- If the history varies with time, specify the solution history as a function handle whose syntax is `y = history(t)`. This function returns an `n`-by-`1` vector that approximates the solution, $y(t)$, for $t <= t_0$. The length of this vector, `n`, is the number of equations in the system you want to solve.
- If $y(t)$ is constant, you can specify `history` as an `n`-by-`1` vector of the constant values.
- If you are calling `ddensd` to continue a previous integration to t_0, you can specify history as the output, `sol`, from the previous integration.
- If you are solving an initial value DDE, specify history as a cell array, `{y0, yp0}`. The first element, `y0`, is a column vector of initial values, $y(t_0)$. The second element, yp_0, is a column vector whose elements are the initial derivatives, $y'(t_0)$. These vectors must be consistent, meaning that they satisfy Equation 1-1 at t_0. See "Initial Value Neutral Delay Differential Equations" on page 1-1257 for more information.

Data Types
`function_handle | single | double | struct | cell`

tspan - Interval of integration

1-by-2 vector

Interval of integration, specified as the vector `[t0 tf]`. The first element, `t0`, is the initial value of t. The second element, `tf`, is the final value of t. The value of `t0` must be less than `tf`.

Data Types
`single | double`

options - Optional integration parameters

structure returned by `ddeset`

Optional integration parameters, specified as a structure created and returned by the `ddeset` function. Some commonly used properties are: `'RelTol'`, `'AbsTol'`, and `'Events'`. See the `ddeset` reference page for more information about specifying `options`.

Output Arguments

sol - Solution

structure

Solution, returned as a structure containing the following fields.

`sol.x`	Mesh selected by `ddensd`.
`sol.y`	An approximation to $y(t)$ at the mesh points.
`sol.yp`	An approximation to $y'(t)$ at the mesh points.
`sol.solver`	A string identifying the solver, `'ddensd'`.

You can pass `sol` to the `deval` function to evaluate the solution at specific points. For example, `y = deval(sol, 0.5*(sol.x(1) + sol.x(end)))` evaluates the solution at the midpoint of the interval of integration.

Definitions

Initial Value Neutral Delay Differential Equations

An initial value DDE has $dy_i \geq t_0$ and $dyp_j \geq t_0$, for all i and j. At $t = t_0$, all delayed terms reduce to $y(dy_i) = y(t_0)$ and $y'(dyp_j) = y'(t_0)$:

$$y'(t_0) = f(t_0, y(t_0), y(t_0),..., y(t_0), y'(t_0),..., y'(t_0)) \quad \textbf{(1-2)}$$

For $t > t_0$, all derivative delays must satisfy $dyp < t$.

When you solve initial value neutral DDEs, you must supply $y'(t_0)$ to `ddensd`. To do this, specify `history` as a cell array `{Y0,YP0}`. Here, `Y0` is the column vector of initial values, $y(t_0)$, and `YP0` is a column vector of initial derivatives, $y'(t_0)$. These vectors must be consistent, meaning that they satisfy Equation 1-2 at t_0.

Examples

Neutral DDE with Two Delays

Solve the following neutral DDE, presented by Paul [1], for $0 \le t \le \pi$:

$$y'(t) = 1 + y(t) - 2y(t/2)^2 - y'(t - \pi)$$

with history: $y(t) = \cos(t)$ for $t \le 0$.

Create a new program file in the editor. This file will contain a main function and four local functions.

Define the first-order DDE as a local function.

```
function yp = ddefun(t,y,ydel,ypdel)
    yp = 1 + y - 2*ydel^2 - ypdel;
end
```

Define the solution delay as a local function.

```
function dy = dely(t,y)
    dy = t/2;
end
```

Define the derivative delay as a local function.

```
function dyp = delyp(t,y)
    dyp = t-pi;
end
```

Define the solution history as a local function.

```
function y = history(t)
    y = cos(t);
end
```

Define the interval of integration and solve the DDE using the `ddensd` function. Add this code to the main function.

```
tspan = [0 pi];
sol = ddensd(@ddefun,@dely,@delyp,@history,tspan);
```

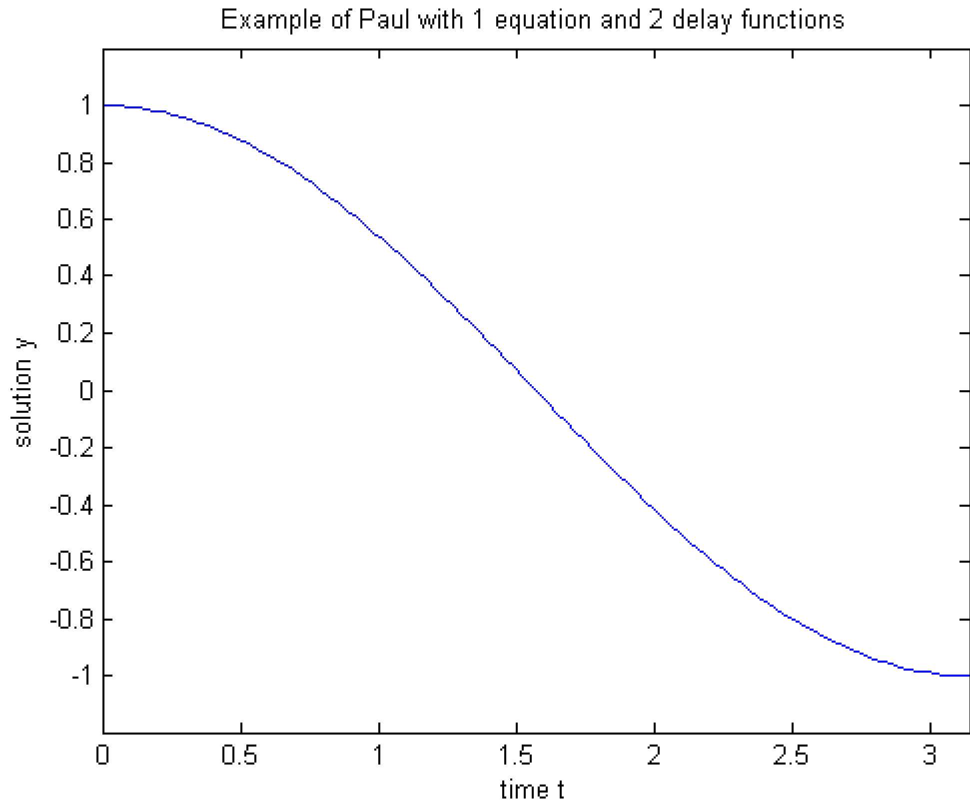
Evaluate the solution at 100 equally spaced points between 0 and π. Add this code to the main function.

```
tn = linspace(0,pi);
yn = deval(sol,tn);
```

Plot the results. Add this code to the main function.

```
plot(tn,yn);
xlim([0 pi]);
ylim([-1.2 1.2]);
xlabel('time t');
ylabel('solution y');
```

Run your program to calculate the solution and display the plot.



Note The file, `ddex4.m`, contains the complete code for this example. To see the code in an editor, type `edit ddex4` at the command line.

Algorithms

For information about the algorithm used in this solver, see Shampine [2].

References

[1] Paul, C.A.H. “A Test Set of Functional Differential Equations.” *Numerical Analysis Reports*. No. 243. Manchester, UK: Math Department, University of Manchester, 1994.

[2] Shampine, L.F. “Dissipative Approximations to Neutral DDEs.” *Applied Mathematics & Computation*. Vol. 203, Number 2, 2008, pp. 641–648.

See Also

`deval` | `ddeset` | `ddesd` | `dde23` | `function_handle`

ddesd

Purpose

Solve delay differential equations (DDEs) with general delays

Syntax

```
sol = ddesd(ddefun,delays,history,tspan)
sol = ddesd(ddefun,delays,history,tspan,options)
```

Arguments

`ddefun`

Function handle that evaluates the right side of the differential equations $y'(t) = f(t,y(t),y(d(1),...,y(d(k)))$. The function must have the form

```
dydt = ddefun(t,y,Z)
```

where `t` corresponds to the current t, `y` is a column vector that approximates $y(t)$, and `Z(:,j)` approximates $y(d(j))$ for delay $d(j)$ given as component j of `delays(t,y)`. The output is a column vector corresponding to $f(t,y(t),y(d(1),...,y(d(k)))$.

`delays`

Function handle that returns a column vector of delays $d(j)$. The delays can depend on both t and $y(t)$. `ddesd` imposes the requirement that $d(j) \leq t$ by using `min`$(d(j),t)$.

If all the delay functions have the form $d(j) = t - \tau_j$, you can set the argument `delays` to a constant vector `delays`(j) = τ_j. With delay functions of this form, `ddesd` is used exactly like `dde23`.

`history`	Specify `history` in one of three ways: • A function of t such that `y = history(t)` returns the solution $y(t)$ for $t \le t_0$ as a column vector • A constant column vector, if $y(t)$ is constant • The solution `sol` from a previous integration, if this call continues that integration
`tspan`	Interval of integration from `t0=tspan(1)` to `tf=tspan(end)` with `t0 < tf`.
`options`	Optional integration argument. A structure you create using the `ddeset` function. See `ddeset` for details.

Description

`sol = ddesd(ddefun,delays,history,tspan)` integrates the system of DDEs

$$y'(t) = f\left(t, y(t), y(d(1)), \ldots, y(d(k))\right)$$

on the interval $[t_0, t_f]$, where delays $d(j)$ can depend on both t and $y(t)$, and $t_0 < t_f$. Inputs `ddefun` and `delays` are function handles. See the `function_handle` reference page for more information.

"Parameterizing Functions" explains how to provide additional parameters to the functions `ddefun`, `delays`, and `history`, if necessary.

`ddesd` returns the solution as a structure `sol`. Use the auxiliary function `deval` and the output `sol` to evaluate the solution at specific points `tint` in the interval `tspan = [t0,tf]`.

```
yint = deval(sol,tint)
```

The structure `sol` returned by `ddesd` has the following fields.

`sol.x`	Mesh selected by `ddesd`
`sol.y`	Approximation to $y(x)$ at the mesh points in `sol.x`.
`sol.yp`	Approximation to $y'(x)$ at the mesh points in `sol.x`
`sol.solver`	Solver name, `'ddesd'`

`sol = ddesd(ddefun,delays,history,tspan,options)` solves as above with default integration properties replaced by values in `options`, an argument created with `ddeset`. See `ddeset` and "Types of DDEs" in the MATLAB documentation for details.

Commonly used options are scalar relative error tolerance `'RelTol'` (`1e-3` by default) and vector of absolute error tolerances `'AbsTol'` (all components are `1e-6` by default).

Use the `'Events'` option to specify a function that `ddesd` calls to find where functions $g(t,y(t),y(d(1)),...,y(d(k)))$ vanish. This function must be of the form

```
[value,isterminal,direction] = events(t,y,Z)
```

and contain an event function for each event to be tested. For the `k`th event function in `events`:

- `value(k)` is the value of the `k`th event function.
- `isterminal(k) = 1` if you want the integration to terminate at a zero of this event function and `0` otherwise.
- `direction(k) = 0` if you want `ddesd` to compute all zeros of this event function, `+1` if only zeros where the event function increases, and `-1` if only zeros where the event function decreases.

If you specify the `'Events'` option and events are detected, the output structure `sol` also includes fields:

`sol.xe`	Row vector of locations of all events, i.e., times when an event function vanished
`sol.ye`	Matrix whose columns are the solution values corresponding to times in `sol.xe`
`sol.ie`	Vector containing indices that specify which event occurred at the corresponding time in `sol.xe`

Examples

The equation

```
sol = ddesd(@ddex1de,@ddex1delays,@ddex1hist,[0,5]);
```

solves a DDE on the interval `[0,5]` with delays specified by the function `ddex1delays` and differential equations computed by `ddex1de`. The history is evaluated for $t \leq 0$ by the function `ddex1hist`. The solution is evaluated at 100 equally spaced points in `[0,5]`:

```
tint = linspace(0,5);
yint = deval(sol,tint);
```

and plotted with

```
plot(tint,yint);
```

This problem involves constant delays. The `delay` function has the form

```
function d = ddex1delays(t,y)
%DDEX1DELAYS  Delays for using with DDEX1DE.
d = [ t - 1
      t - 0.2];
```

The problem can also be solved with the syntax corresponding to constant delays

```
delays = [1, 0.2];
sol = ddesd(@ddex1de,delays,@ddex1hist,[0, 5]);
```

or using `dde23`:

```
sol = dde23(@ddex1de,delays,@ddex1hist,[0, 5]);
```

For more examples of solving delay differential equations see `ddex2` and `ddex3`.

References

[1] Shampine, L.F., "Solving ODEs and DDEs with Residual Control," *Applied Numerical Mathematics*, Vol. 52, 2005, pp. 113-127.

See Also

`dde23` | `ddeget` | `ddensd` | `ddeset` | `deval` | `function_handle`

Purpose

Create or alter delay differential equations options structure

Syntax

```
options = ddeset('name1',value1,'name2',value2,...)
options = ddeset(oldopts,'name1',value1,...)
options = ddeset(oldopts,newopts)
ddeset
```

Description

`options = ddeset('name1',value1,'name2',value2,...)` creates an integrator options structure `options` in which the named properties have the specified values. Any unspecified properties have default values. It is sufficient to type only the leading characters that uniquely identify the property. `ddeset` ignores case for property names.

`options = ddeset(oldopts,'name1',value1,...)` alters an existing options structure `oldopts`. This overwrites any values in `oldopts` that are specified using name/value pairs and returns the modified structure as the output argument.

`options = ddeset(oldopts,newopts)` combines an existing options structure `oldopts` with a new options structure `newopts`. Any values set in `newopts` overwrite the corresponding values in `oldopts`.

`ddeset` with no input arguments displays all property names and their possible values, indicating defaults with braces `{}`.

You can use the function `ddeget` to query the `options` structure for the value of a specific property.

DDE Properties

The following sections describe the properties that you can set using `ddeset`. There are several categories of properties:

- Error control
- Solver output
- Step size
- Event location
- Discontinuities

Error Control Properties

At each step, the DDE solvers estimate an error `e`. The `dde23` function estimates the local truncation error, and the other solvers estimate the residual. In either case, this error must be less than or equal to the acceptable error, which is a function of the specified relative tolerance, `RelTol`, and the specified absolute tolerance, `AbsTol`.

```
|e(i)|*max(RelTol*abs(y(i)),AbsTol(i))
```

For routine problems, the solvers deliver accuracy roughly equivalent to the accuracy you request. They deliver less accuracy for problems integrated over "long" intervals and problems that are moderately unstable. Difficult problems may require tighter tolerances than the default values. For relative accuracy, adjust `RelTol`. For the absolute error tolerance, the scaling of the solution components is important: if `|y|` is somewhat smaller than `AbsTol`, the solver is not constrained to obtain any correct digits in `y`. You might have to solve a problem more than once to discover the scale of solution components.

Roughly speaking, this means that you want `RelTol` correct digits in all solution components except those smaller than thresholds `AbsTol(i)`. Even if you are not interested in a component `y(i)` when it is small, you may have to specify `AbsTol(i)` small enough to get some correct digits in `y(i)` so that you can accurately compute more interesting components.

The following table describes the error control properties.

DDE Error Control Properties

<table>
<tr><th>Property</th><th>Value</th><th>Description</th></tr>
<tr><td>RelTol</td><td>Positive scalar {1e-3}</td><td>A relative error tolerance that applies to all components of the solution vector y. It is a measure of the error relative to the size of each solution component. Roughly, it controls the number of correct digits in all solution components except those smaller than thresholds AbsTol(i). The default, 1e-3, corresponds to 0.1% accuracy.

The estimated error in each integration step satisfies |e(i)|max(RelTol*abs(y(i)),AbsTol(i)).</td></tr>
<tr><td>AbsTol</td><td>Positive scalar or vector {1e-6}</td><td>Absolute error tolerances that apply to the individual components of the solution vector. AbsTol(i) is a threshold below which the value of the ith solution component is unimportant. The absolute error tolerances determine the accuracy when the solution approaches zero. Even if you are not interested in a component y(i) when it is small, you may have to specify AbsTol(i) small enough to get some correct digits in y(i) so that you can accurately compute more interesting components.

If AbsTol is a vector, the length of AbsTol must be the same as the length of the solution vector y. If AbsTol is a scalar, the value applies to all components of y.</td></tr>
<tr><td>NormControl</td><td>on | {off}</td><td>Control error relative to norm of solution. Set this property on to request that the solvers control the error in each integration step with norm(e)<= max(RelTol*norm(y),AbsTol). By default, the solvers use a more stringent component-wise error control.</td></tr>
</table>

Solver Output Properties

You can use the solver output properties to control the output that the solvers generate.

DDE Solver Output Properties

Property	Value	Description

OutputFcn	Function handle {@odeplot}	The output function is a function that the solver calls after every successful integration step. To specify an output function, set 'OutputFcn' to a function handle. For example,
		
		`options = ddeset('OutputFcn',...`
		`@myfun)`
		
		sets 'OutputFcn' to @myfun, a handle to the function myfun. See the function_handle reference page for more information.
		
		The output function must be of the form
		
		`status = myfun(t,y,flag)`
		
		"Parameterizing Functions" explains how to provide additional parameters to myfun, if necessary.
		
		The solver calls the specified output function with the following flags. Note that the syntax of the call differs with the flag. The function must respond appropriately:
		
		• init — The solver calls myfun(tspan,y0,'init') before beginning the integration to allow the output function to initialize. tspan is the input argument to the solvers. y0 is the initial value of the solution, either from history(t0) or specified in the initialY option.
		
		• {none} — The solver calls status = myfun(t,y) after each integration step on which output is requested. t contains points where output was

DDE Solver Output Properties (Continued)

Property	Value	Description
		generated during the step, and `y` is the numerical solution at the points in `t`. If `t` is a vector, the `i`th column of `y` corresponds to the `i`th element of `t`. `myfun` must return a `status` output value of `0` or `1`. If literal > `status`, the solver halts integration. You can use this mechanism, for instance, to implement a **Stop** button. • done — The solver calls `myfun([],[],'done')` when integration is complete to allow the output function to perform any cleanup chores. You can use these general purpose output functions or you can edit them to create your own. Type `help functionname` at the command line for more information. • `odeplot` – time series plotting (default when you call the solver with no output argument and you have not specified an output function) • `odephas2` – two-dimensional phase plane plotting • `odephas3` – three-dimensional phase plane plotting • `odeprint` – print solution as the solver computes it

`OutputSel`	Vector of indices	Vector of indices specifying which components of the solution vector the solvers pass to the output function. For example, if you want to use the `odeplot` output function, but you want to plot only the first and third components of the solution, you can do this using
		
		`options = ddeset...`
		`('OutputFcn',@odeplot,...`

DDE Solver Output Properties (Continued)

Property	Value	Description
		`'OutputSel',[1 3]);` By default, the solver passes all components of the solution to the output function.
`Stats`	`on` | `{off}`	Specifies whether the solver should display statistics about its computations. By default, `Stats` is `off`. If it is `on`, after solving the problem the solver displays: • The number of successful steps • The number of failed attempts • The number of times the DDE function was called

Step Size Properties

The step size properties let you specify the size of the first step the solver tries, potentially helping it to better recognize the scale of the problem. In addition, you can specify bounds on the sizes of subsequent time steps.

The following table describes the step size properties.

DDE Step Size Properties

Property	Value	Description
`InitialStep`	Positive scalar	Suggested initial step size. `InitialStep` sets an upper bound on the magnitude of the first step size the solver tries. If you do not set `InitialStep`, the solver bases the initial step size on the slope of the solution at the initial time `tspan(1)`. The initial step size is limited by the shortest delay. If the slope of all solution components is zero, the procedure might try a step size that is much too large. If you know

DDE Step Size Properties (Continued)

Property	Value	Description
		this is happening or you want to be sure that the solver resolves important behavior at the start of the integration, help the code start by providing a suitable `InitialStep`.
`MaxStep`	Positive scalar {`0.1* abs(t0-tf)`}	Upper bound on solver step size. If the differential equation has periodic coefficients or solutions, it may be a good idea to set `MaxStep` to some fraction (such as 1/4) of the period. This guarantees that the solver does not enlarge the time step too much and step over a period of interest. Do *not* reduce `MaxStep`: • When the solution does not appear to be accurate enough. Instead, reduce the relative error tolerance `RelTol`, and use the solution you just computed to determine appropriate values for the absolute error tolerance vector `AbsTol`. (See "Error Control Properties" on page 1-1268 for a description of the error tolerance properties.) • To make sure that the solver doesn't step over some behavior that occurs only once during the simulation interval. If you know the time at which the change occurs, break the simulation interval into two pieces and call the solver twice. If you do not know the time at which the change occurs, try reducing the error tolerances `RelTol` and `AbsTol`. Use `MaxStep` as a last resort.

Event Location Property

In some DDE problems, the times of specific events are important. While solving a problem, the solvers can detect such events by locating transitions to, from, or through zeros of user-defined functions.

The following table describes the `Events` property.

DDE Events Property

String	Value	Description

`Events`	Function handle	A function handle that includes one or more event functions. For `dde23` and `ddesd`, this function has the following syntax:
		`[value,isterminal,direction] = events(t,y,YDEL)`
		For `ddensd`, the syntax is:
		`[value,isterminal,direction] = events(t,y,YDEL,YPDEL)`
		The output arguments, `value`, `isterminal`, and `direction`, are vectors for which the `i`th element corresponds to the `i`th event function:
		• `value(i)` is the value of the `i`th event function.
		• `isterminal(i) = 1` if you want the integration to terminate at a zero of this event function, and `0` otherwise.
		• `direction(i) = 0` if you want the solver to locate all zeros (the default), `+1` if only zeros where the event function is increasing, and `-1` if only zeros where the event function is decreasing.
		If you specify an events function and events are detected, the solver returns three additional fields in the solution structure `sol`:
		• `sol.xe` is a row vector of times at which events occur.
		• `sol.ye` is a matrix whose columns are the solution values corresponding to times in `sol.xe`.
		• `sol.ie` is a vector containing indices that specify which event occurred at the corresponding time in `sol.xe`.

DDE Events Property (Continued)

String	Value	Description
		For examples that use an event function while solving ordinary differential equation problems, see “Event Location” (`ballode`) and “Advanced Event Location” (`orbitode`), in the MATLAB Mathematics documentation.

Discontinuity Properties

The solver functions can solve problems with discontinuities in the history or in the coefficients of the equations. The following properties enable you to provide these solvers with a different initial value, and, for `dde23`, locations of known discontinuities. For more information, see “Discontinuities in DDEs”.

The following table describes the discontinuity properties.

DDE Discontinuity Properties

String	Value	Description
`Jumps`	Vector	Location of discontinuities. Points t where the history or solution may have a jump discontinuity in a low-order derivative. This applies only to the `dde23` solver.
`InitialY`	Vector	Initial value of solution. By default the initial value of the solution is the value returned by `history` at the initial point. Supply a different initial value as the value of the `InitialY` property.

Examples

To create an options structure that changes the relative error tolerance of the solver from the default value of `1e-3` to `1e-4`, enter

```
options = ddeset('RelTol',1e-4);
```

To recover the value of 'RelTol' from options, enter

```
ddeget(options,'RelTol')

ans =

  1.0000e-004
```

See Also

dde23 | ddensd | ddesd | ddeget | function_handle

Purpose

Distribute inputs to outputs

Note Beginning with MATLAB Version 7.0 software, you can access the contents of cell arrays and structure fields without using the `deal` function. See Example 3, below.

Syntax

```
[Y1, Y2, Y3, ...] = deal(X)
[Y1, Y2, Y3, ...] = deal(X1, X2, X3, ...)
[S.field] = deal(X)
[X{:}] = deal(A.field)
[Y1, Y2, Y3, ...] = deal(X{:})
[Y1, Y2, Y3, ...] = deal(S.field)
```

Description

`[Y1, Y2, Y3, ...]  = deal(X)` copies the single input to all the requested outputs. It is the same as `Y1 = X`, `Y2 = X`, `Y3 = X`, `...`

`[Y1, Y2, Y3, ...]  = deal(X1, X2, X3, ...)` is the same as `Y1 = X1`; `Y2 = X2`; `Y3 = X3`; `...`

Tips

`deal` is most useful when used with cell arrays and structures via comma-separated list expansion. Here are some useful constructions:

`[S.field] = deal(X)` sets all the fields with the name `field` in the structure array `S` to the value `X`. If `S` doesn't exist, use `[S(1:m).field] = deal(X)`.

`[X{:}] = deal(A.field)` copies the values of the field with name `field` to the cell array `X`. If `X` doesn't exist, use `[X{1:m}] = deal(A.field)`.

`[Y1, Y2, Y3, ...]  = deal(X{:})` copies the contents of the cell array `X` to the separate variables `Y1, Y2, Y3, ...`

`[Y1, Y2, Y3, ...]  = deal(S.field)` copies the contents of the fields with the name `field` to separate variables `Y1, Y2, Y3, ...`

Examples

Example 1 — Assign Data From a Cell Array

Use `deal` to copy the contents of a 4-element cell array into four separate output variables.

```
C = {rand(3) ones(3,1) eye(3) zeros(3,1)};
[a,b,c,d] = deal(C{:})

a =
    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214

b =
    1
    1
    1

c =
    1    0    0
    0    1    0
    0    0    1

d =
    0
    0
    0
```

Example 2 — Assign Data From Structure Fields

Use `deal` to obtain the contents of all the name fields in a structure array:

```
A.name = 'Pat'; A.number = 176554;
A(2).name = 'Tony'; A(2).number = 901325;
[name1,name2] = deal(A(:).name)

name1 =
   Pat
```

```
name2 =
   Tony
```

Example 3 — Doing the Same Without deal

Beginning with MATLAB Version 7.0 software, you can, in most cases, access the contents of cell arrays and structure fields without using the `deal` function. The two commands shown below perform the same operation as those used in the previous two examples, except that these commands do not require `deal`.

```
[a,b,c,d] = C{:}
[name1,name2] = A(:).name
```

See Also

`cell` | `iscell` | `celldisp` | `struct` | `isstruct` | `fieldnames` | `isfield` | `orderfields` | `rmfield` | `cell2struct` | `struct2cell`

deblank

Purpose

Strip trailing blanks from end of string

Syntax

```
str = deblank(str)
c = deblank(c)
```

Description

`str = deblank(str)` removes all trailing whitespace and null characters from the end of character string `str`. A whitespace is any character for which the `isspace` function returns logical 1 (`true`).

`c = deblank(c)` when `c` is a cell array of strings, applies `deblank` to each element of `c`.

The `deblank` function is useful for cleaning up the rows of a character array.

Examples

Example 1 – Removing Trailing Blanks From a String

Compose a string `str` that contains space, tab, and null characters:

```
NL = char(0);    TAB = char(9);
str = [NL 32 TAB NL 'AB' 32 NL 'CD' NL 32 TAB NL 32];
```

Display all characters of the string between | symbols:

```
['|' str '|']
ans =
    |    AB  CD       |
```

Remove trailing whitespace and null characters, and redisplay the string:

```
newstr = deblank(str);

['|' newstr '|']
ans =
    |    AB  CD|
```

Example 2– Removing Trailing Blanks From a Cell Array of Strings

Create a 2-by-2 cell array in which each cell contains a word with trailing blanks:

```
A{1,1} = 'MATLAB    ';
A{1,2} = 'SIMULINK    ';
A{2,1} = 'Toolboxes    ';
A{2,2} = 'MathWorks    '
A =
    'MATLAB    '        'SIMULINK    '
    'Toolboxes    '    'MathWorks    '
```

Remove the trailing blanks and redisplay the cell array:

```
deblank(A);
A
A =
    'MATLAB'       'SIMULINK'
    'Toolboxes'    'MathWorks'
```

See Also

strjust | strtrim

Purpose

Convert decimal to base N number in string

Syntax

```
str = dec2base(d, base)
str = dec2base(d, base, n)
```

Description

`str = dec2base(d, base)` converts the nonnegative integer `d` to the specified `base`. `d` must be a nonnegative integer smaller than 2^52, and `base` must be an integer between 2 and 36. The returned argument `str` is a string.

`str = dec2base(d, base, n)` produces a representation with at least `n` digits.

Examples

The expression `dec2base(23, 2)` converts 23_{10} to base 2, returning the string `'10111'`.

See Also

`base2dec`

Purpose

Convert decimal to binary number in string

Syntax

```
str = dec2bin(d)
str = dec2bin(d,n)
```

Description

returns the

`str = dec2bin(d)` binary representation of `d` as a string. `d` must be a nonnegative integer smaller than 2^52.

`str = dec2bin(d,n)` produces a binary representation with at least `n` bits.

The output of `dec2bin` is independent of the endian settings of the computer you are using.

Examples

Decimal 23 converts to binary 010111:

```
dec2bin(23)
ans =
    10111
```

See Also

`bin2dec` | `dec2hex`

dec2hex

Purpose

Convert decimal to hexadecimal number in string

Syntax

```
str = dec2hex(d)
str = dec2hex(d, n)
```

Description

`str = dec2hex(d)` converts the decimal integer `d` to its hexadecimal representation stored in a MATLAB string. `d` must be a nonnegative integer smaller than 2^52. MATLAB converts noninteger inputs, such as those of class `double` or `char`, to their integer equivalents before converting to hexadecimal.

`str = dec2hex(d, n)` produces a hexadecimal representation with at least `n` digits.

Examples

To convert decimal 1023 to hexadecimal,

```
dec2hex(1023)
ans =
    3FF

dec2hex(1023, 6)
ans =
0003FF
```

Convert 2-by-5 array `A` to hexadecimal:

```
A = [3487,    125,  8997,  1433,  189; ...
      771,  84832,   118,  9366,  212];

 A(:)                        dec2hex(A)
ans =                        ans =
     3487                       00D9F
      771                       00303
      125                       0007D
    84832                       14B60
     8997                       02325
      118                       00076
     1433                       00599
```

```
    9366                  02496
     189                  000BD
     212                  000D4
```

See Also

dec2bin | format | hex2dec | hex2num

Purpose

Compute consistent initial conditions for ode15i

Syntax

```
[y0mod,yp0mod] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0)
[y0mod,yp0mod] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0,
   options)
[y0mod,yp0mod,resnrm] = decic(odefun,t0,y0,fixed_y0,yp0,
   fixed_yp0...)
```

Description

`[y0mod,yp0mod] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0)` uses the inputs `y0` and `yp0` as initial guesses for an iteration to find output values that satisfy the requirement $f(t0, y0\,\mathrm{mod}, yp0\,\mathrm{mod}) = 0$, i.e., `y0mod` and `yp0mod` are consistent initial conditions. `odefun` is a function handle. The function `decic` changes as few components of the guesses as possible. You can specify that `decic` holds certain components fixed by setting `fixed_y0(i) = 1` if no change is permitted in the guess for `y0(i)` and 0 otherwise. `decic` interprets `fixed_y0 = []` as allowing changes in all entries. `fixed_yp0` is handled similarly.

"Parameterizing Functions" explains how to provide additional parameters to the function `odefun`, if necessary.

You cannot fix more than `length(y0)` components. Depending on the problem, it may not be possible to fix this many. It also may not be possible to fix certain components of `y0` or `yp0`. It is recommended that you fix no more components than necessary.

`[y0mod,yp0mod] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0,options)` computes as above with default tolerances for consistent initial conditions, `AbsTol` and `RelTol`, replaced by the values in options, a structure you create with the `odeset` function.

`[y0mod,yp0mod,resnrm] = decic(odefun,t0,y0,fixed_y0,yp0,fixed_yp0...)` returns the norm of `odefun(t0,y0mod,yp0mod)` as `resnrm`. If the norm seems unduly large, use `options` to decrease `RelTol` (`1e-3` by default).

Examples

The files, `ihb1dae.m` and `iburgersode.m`, provide examples which use `decic` to solve implicit ODEs.

See Also

`ode15i` | `odeget` | `odeset` | `function_handle`

deconv

Purpose

Deconvolution and polynomial division

Syntax

```
[q,r] = deconv(v,u)
```

Description

`[q,r] = deconv(v,u)` deconvolves vector `u` out of vector `v`, using long division. The quotient is returned in vector `q` and the remainder in vector `r` such that `v = conv(u,q)+r` .

If `u` and `v` are vectors of polynomial coefficients, convolving them is equivalent to multiplying the two polynomials, and deconvolution is polynomial division. The result of dividing `v` by `u` is quotient `q` and remainder `r`.

Examples

If

```
u = [1   2   3   4]
v = [10   20   30]
```

the convolution is

```
c = conv(u,v)
c =
    10     40     100     160     170     120
```

Use deconvolution to recover `u`:

```
[q,r] = deconv(c,u)
q =
    10    20    30
r =
    0     0     0     0     0     0
```

This gives a quotient equal to `v` and a zero remainder.

Algorithms

`deconv` uses the `filter` primitive.

See Also

`conv` | `residue`

Purpose

Discrete Laplacian

Syntax

```
L = del2(U)
L = del2(U,h)
L = del2(U,hx,hy)
L = del2(U,hx,hy,hz,...)
```

Definitions

If the matrix `U` is regarded as a function *u(x,y)* evaluated at the point on a square grid, then `4*del2(U)` is a finite difference approximation of Laplace's differential operator applied to u, that is:

$$l = \frac{\nabla^2 u}{4} = \frac{1}{4}\left(\frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2}\right)$$

where:

$$l_{ij} = \frac{1}{4}\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}\right) - u_{i,j}$$

in the interior. On the edges, the same formula is applied to a cubic extrapolation.

For functions of more variables *u(x,y,z,...)*, `del2(U)` is an approximation,

$$l = \frac{\nabla^2 u}{2N} = \frac{1}{2N}\left(\frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2} + \frac{d^2 u}{dz^2} + \ldots\right)$$

where N is the number of variables in u.

Description

`L = del2(U)` where `U` is a rectangular array is a discrete approximation of

$$l = \frac{\nabla^2 u}{4} = \frac{1}{4}\left(\frac{d^2 u}{dx^2} + \frac{d^2 u}{dy^2}\right)$$

The matrix `L` is the same size as `U` with each element equal to the difference between an element of `U` and the average of its four neighbors.

`L = del2(U)` when `U` is an multidimensional array, returns an approximation of

$$\frac{\nabla^2 u}{2N}$$

where N is `ndims(u)`.

`L = del2(U,h)` where `H` is a scalar uses `H` as the spacing between points in each direction (`h=1` by default).

`L = del2(U,hx,hy)` when `U` is a rectangular array, uses the spacing specified by `hx` and `hy`. If `hx` is a scalar, it gives the spacing between points in the x-direction. If `hx` is a vector, it must be of length `size(u,2)` and specifies the x-coordinates of the points. Similarly, if `hy` is a scalar, it gives the spacing between points in the y-direction. If `hy` is a vector, it must be of length `size(u,1)` and specifies the y-coordinates of the points.

`L = del2(U,hx,hy,hz,...)` where `U` is multidimensional uses the spacing given by `hx`, `hy`, `hz`, ...

Tips

MATLAB software computes the boundaries of the grid by extrapolating the second differences from the interior. The algorithm used for this computation can be seen in the `del2` program file code. To view this code, type:

```
type del2
```

Examples

The function:

$$u(x, y) = x^2 + y^2$$

has:

$$\nabla^2 u = 4$$

For this function, `4*del2(U)` is also `4`.

```
[x,y] = meshgrid(-4:4,-3:3);
U = x.*x+y.*y
U =
    25    18    13    10     9    10    13    18    25
    20    13     8     5     4     5     8    13    20
    17    10     5     2     1     2     5    10    17
    16     9     4     1     0     1     4     9    16
    17    10     5     2     1     2     5    10    17
    20    13     8     5     4     5     8    13    20
    25    18    13    10     9    10    13    18    25

V = 4*del2(U)
V =
     4     4     4     4     4     4     4     4     4
     4     4     4     4     4     4     4     4     4
     4     4     4     4     4     4     4     4     4
     4     4     4     4     4     4     4     4     4
     4     4     4     4     4     4     4     4     4
     4     4     4     4     4     4     4     4     4
     4     4     4     4     4     4     4     4     4
```

See Also

`diff` | `gradient`

DelaunayTri

Superclasses

TriRep

Purpose

(Will be removed) Delaunay triangulation in 2-D and 3-D

Note `DelaunayTri` will be removed in a future release. Use `delaunayTriangulation` instead.

Description

`DelaunayTri` creates a Delaunay triangulation object from a set of points. You can incrementally modify the triangulation by adding or removing points. In 2-D triangulations you can impose edge constraints. You can perform topological and geometric queries, and compute the Voronoi diagram and convex hull.

Definitions

The 2-D Delaunay triangulation of a set of points is the triangulation in which no point of the set is contained in the circumcircle for any triangle in the triangulation. The definition extends naturally to higher dimensions.

Construction

DelaunayTri	(Will be removed) Construct Delaunay triangulation

Methods

convexHull	(Will be removed) Convex hull
inOutStatus	(Will be removed) Status of triangles in 2-D constrained Delaunay triangulation
nearestNeighbor	(Will be removed) Point closest to specified location

pointLocation	(Will be removed) Simplex containing specified location
voronoiDiagram	(Will be removed) Voronoi diagram

Inherited methods

baryToCart	(Will be removed) Convert point coordinates from barycentric to Cartesian
cartToBary	(Will be removed) Convert point coordinates from cartesian to barycentric
circumcenters	(Will be removed) Circumcenters of specified simplices
edgeAttachments	(Will be removed) Simplices attached to specified edges
edges	(Will be removed) Triangulation edges
faceNormals	(Will be removed) Unit normals to specified triangles
featureEdges	(Will be removed) Sharp edges of surface triangulation
freeBoundary	(Will be removed) Facets referenced by only one simplex
incenters	(Will be removed) Incenters of specified simplices
isEdge	(Will be removed) Test if vertices are joined by edge
neighbors	(Will be removed) Simplex neighbor information

size	(Will be removed) Size of triangulation matrix
vertexAttachments	(Will be removed) Return simplices attached to specified vertices

Properties

`Constraints`	`Constraints` is a `numc`-by-2 matrix that defines the constrained edge data in the triangulation, where `numc` is the number of constrained edges. Each constrained edge is defined in terms of its endpoint indices into `X`. The constraints can be specified when the triangulation is constructed or can be imposed afterwards by directly editing the constraints data. This feature is only supported for 2-D triangulations.
`X`	The dimension of `X` is `mpts`-by-`ndim`, where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside. If column vectors of `x`,`y` or `x`,`y`,`z` coordinates are used to construct the triangulation, the data is consolidated into a single matrix `X`.
`Triangulation`	Triangulation is a matrix representing the set of simplices (triangles or tetrahedra etc.) that make up the triangulation. The matrix is of size `mtri`-by-`nv`, where `mtri` is the number of simplices and `nv` is the number of vertices per simplex. The triangulation is represented by standard simplex-vertex format; each row specifies a simplex defined by indices into `X`, where `X` is the array of point coordinates.

Instance Hierarchy

`DelaunayTri` is a subclass of `TriRep`.

Copy Semantics

Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

See Also

`scatteredInterpolant` | `triangulation` | `delaunayTriangulation`

DelaunayTri

Purpose

(Will be removed) Construct Delaunay triangulation

Note `DelaunayTri` will be removed in a future release. Use `delaunayTriangulation` instead.

Syntax

```
DT = DelaunayTri()
DT = DelaunayTri(X)
DT = DelaunayTri(x,y)
DT = DelaunayTri(x,y,z)
DT = DelaunayTri(..., C)
```

Description

`DT = DelaunayTri()` creates an empty Delaunay triangulation.

`DT = DelaunayTri(X)`, `DT = DelaunayTri(x,y)` and `DT = DelaunayTri(x,y,z)` create a Delaunay triangulation from a set of points. The points can be specified as an `mpts`-by-`ndim` matrix `X`, where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside, where `ndim` is 2 or 3. Alternatively, the points can be specified as column vectors `(x,y)` or `(x,y,z)` for 2-D and 3-D input.

`DT = DelaunayTri(..., C)` creates a constrained Delaunay triangulation. The edge constraints `C` are defined by an `numc`-by-2 matrix, `numc` being the number of constrained edges. Each row of `C` defines a constrained edge in terms of its endpoint indices into the point set `X`. This feature is only supported for 2-D triangulations.

Definitions

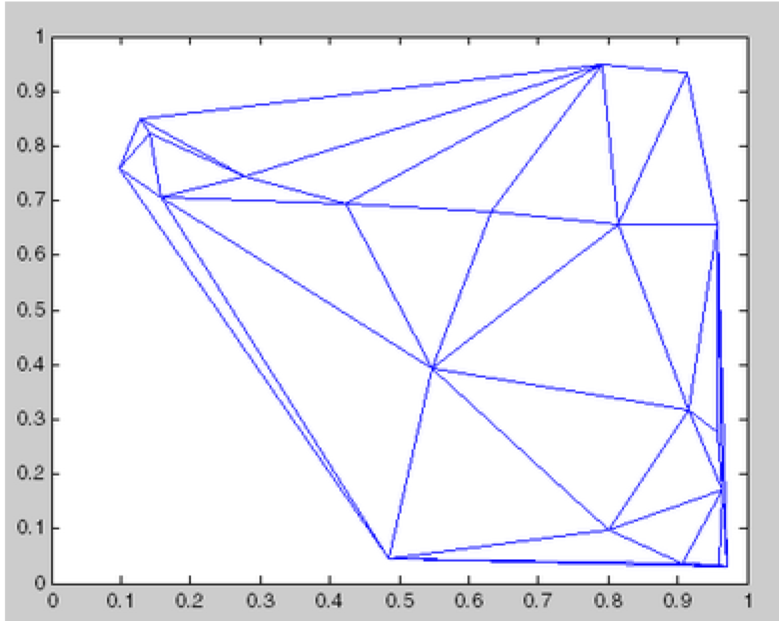
The 2-D Delaunay triangulation of a set of points is the triangulation in which no point of the set is contained in the circumcircle for any triangle in the triangulation. The definition extends naturally to higher dimensions.

Examples

Compute the Delaunay triangulation of twenty random points located within a unit square.

```
x = rand(20,1);
```

```
y = rand(20,1);
dt = DelaunayTri(x,y)
triplot(dt);
```



For more examples, type `help demoDelaunayTri` at the MATLAB command-line prompt.

See Also

`scatteredInterpolant` | `triangulation` | `delaunayTriangulation`

delaunay

Purpose

Delaunay triangulation

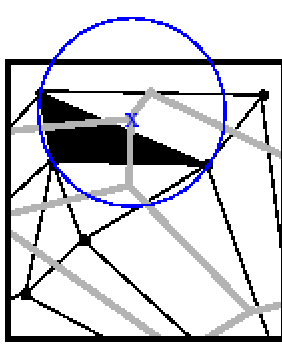
Note Qhull-specific options are no longer supported. Remove the `OPTIONS` argument from all instances in your code that pass it to `delaunay`.

Syntax

```
TRI = delaunay(X,Y)
TRI = delaunay(X,Y,Z)
TRI = delaunay(X)
```

Definitions

`delaunay` creates a Delaunay triangulation of a set of points in 2-D or 3-D space. A 2-D Delaunay triangulation ensures that the circumcircle associated with each triangle contains no other point in its interior. This definition extends naturally to higher dimensions.



Description

`TRI = delaunay(X,Y)`creates a 2-D Delaunay triangulation of the points (`X`,`Y`), where `X` and `Y` are column-vectors. `TRI` is a matrix representing the set of triangles that make up the triangulation. The matrix is of size `mtri`-by-3, where `mtri` is the number of triangles. Each row of `TRI` specifies a triangle defined by indices with respect to the points.

`TRI = delaunay(X,Y,Z)` creates a 3-D Delaunay triangulation of the points (`X`,`Y`,`Z`), where `X`, `Y`, and `Z` are column-vectors. `TRI` is a matrix

representing the set of tetrahedra that make up the triangulation. The matrix is of size mtri-by-4, where mtri is the number of tetrahedra. Each row of TRI specifies a tetrahedron defined by indices with respect to the points.

TRI = delaunay(X) creates a 2-D or 3-D Delaunay triangulation from the point coordinates X. This variant supports the definition of points in matrix format. X is of size mpts-by-ndim, where mpts is the number of points and ndim is the dimension of the space where the points reside, $2 \leqq$ ndim $\leqq 3$. The output triangulation is equivalent to that of the dedicated functions supporting the 2-input or 3-input calling syntax.

delaunay produces an isolated triangulation, useful for applications like plotting surfaces via the trisurf function. If you wish to query the triangulation; for example, to perform nearest neighbor, point location, or topology queries, use delaunayTriangulation instead.

Visualization

Use one of these functions to plot the output of delaunay:

triplot	Displays the triangles defined in the m-by-3 matrix TRI.
trisurf	Displays each triangle defined in the m-by-3 matrix TRI as a surface in 3-D space. To see a 2-D surface, you can supply a vector of some constant value for the third dimension. For example `trisurf(TRI,x,y,zeros(size(x)))`

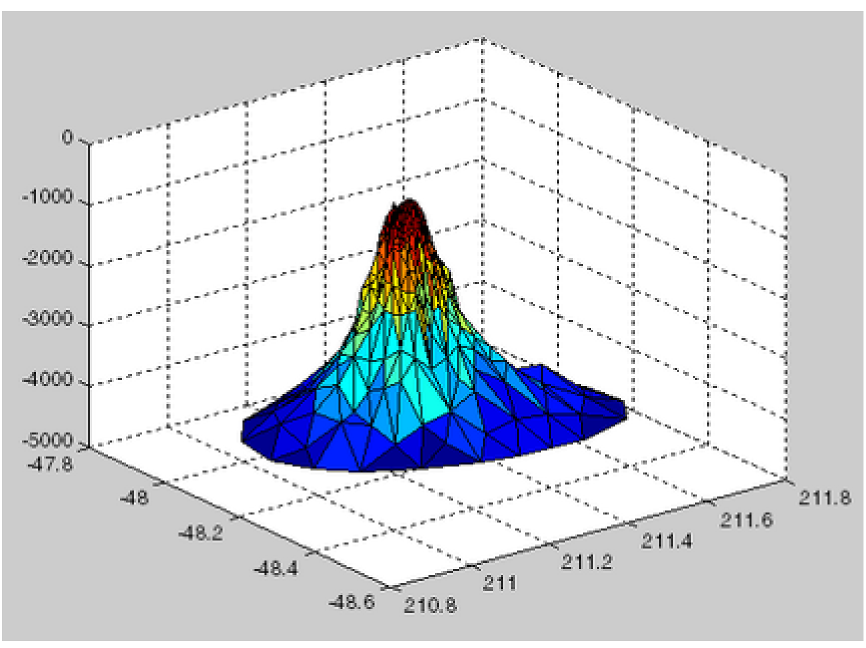
`trimesh`	Displays each triangle defined in the m-by-3 matrix TRI as a mesh in 3-D space. To see a 2-D surface, you can supply a vector of some constant value for the third dimension. For example, `trimesh(TRI,x,y,zeros(size(x)))` produces almost the same result as `triplot`, except in 3-D space.
`tetramesh`	Plots a triangulation composed of tetrahedra.

Examples

Example

Plot the Delaunay triangulation of a large dataset:

```
load seamount
tri = delaunay(x,y);
trisurf(tri,x,y,z);
```



See Also

`delaunayTriangulation` | `scatteredInterpolant` | `plot` | `triplot` | `trimesh` | `trisurf`

delaunayn

Purpose

N-D Delaunay triangulation

Syntax

```
T = delaunayn(X)
T = delaunayn(X, options)
```

Description

`T = delaunayn(X)` computes a set of simplices such that no data points of `X` are contained in any circumspheres of the simplices. The set of simplices forms the Delaunay triangulation. `X` is an `m`-by-`n` array representing m points in n-dimensional space. `T` is a `numt`-by-(`n+1`) array where each row contains the indices into `X` of the vertices of the corresponding simplex.

`T = delaunayn(X, options)` specifies a cell array of strings `options`. The default options are:

- `{'Qt','Qbb','Qc'}` for 2- and 3-dimensional input
- `{'Qt','Qbb','Qc','Qx'}` for 4 and higher-dimensional input

If `options` is `[]`, the default options used. If `options` is `{''}`, no options are used, not even the default.

Visualization

Plotting the output of `delaunayn` depends of the value of `n`:

- For `n = 2`, use `triplot`, `trisurf`, or `trimesh` as you would for `delaunay`.
- For `n = 3`, use `tetramesh`.

  For more control over the color of the facets, use `patch` to plot the output.
- You cannot plot `delaunayn` output for `n > 3`.

Examples

Example 1

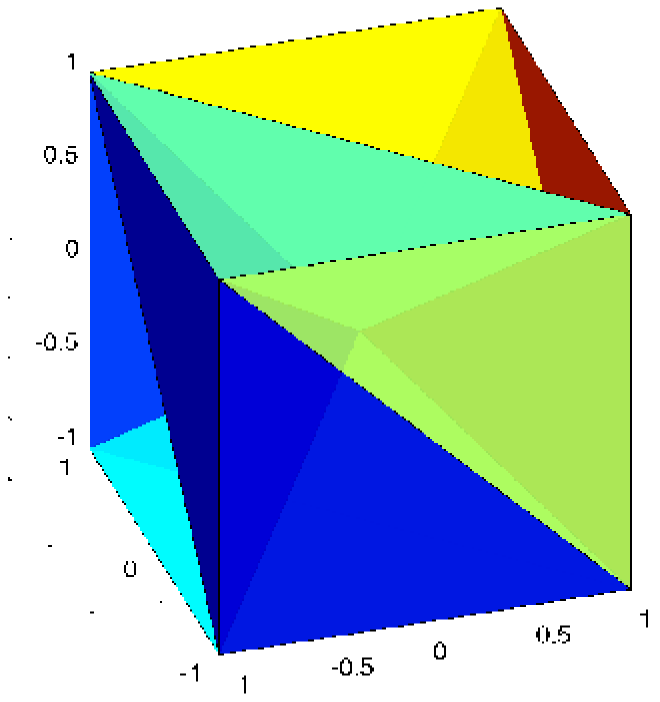
This example generates an n-dimensional Delaunay triangulation, where `n = 3`.

```
d = [-1 1];
[x,y,z] = meshgrid(d,d,d);  % A cube
```

```
x = [x(:);0];
y = [y(:);0];
z = [z(:);0];
% [x,y,z] are corners of a cube plus the center.
X = [x(:) y(:) z(:)];
Tes = delaunayn(X)

Tes =

     4     3     9     1
     4     9     2     1
     7     9     3     1
     7     5     9     1
     7     9     4     3
     7     8     4     9
     6     2     9     1
     6     9     5     1
     6     4     9     2
     6     4     8     9
     6     9     7     5
     6     8     7     9
```

You can use `tetramesh` to visualize the tetrahedrons that form the corresponding simplex. `camorbit` rotates the camera position to provide a meaningful view of the figure.

```
tetramesh(Tes,X);camorbit(20,0)
```



See Also delaunayTriangulation | convhulln | tetramesh | voronoin

Superclasses triangulation

Purpose Delaunay triangulation in 2-D and 3-D

Description

Use the `delaunayTriangulation` class to create a 2-D or 3-D triangulation from a set of points. When your points are in 2-D, you can specify edge constraints .

You can perform a variety of topological and geometric queries on a `delaunayTriangulation`, including any `triangulation` query. For example, locate a facet that contains a specific point, find the vertices of the convex hull, or compute the Voronoi Diagram.

Construction

`DT = delaunayTriangulation(P)` creates a Delaunay triangulation from the points in `P`. Matrix `P` has 2 or 3 columns, depending on whether your points are in 2-D or 3-D space.

`DT = delaunayTriangulation(P,C)` specifies the edge constraints in matrix `C`. In this case, `P` specifies points in 2-D. Each row of `C` defines the start and end vertex IDs of a constrained edge.

`DT = delaunayTriangulation(x,y)` creates a 2-D Delaunay triangulation from the point coordinates in the column vectors, `x` and `y`.

`DT = delaunayTriangulation(x,y,C)` specifies the edge constraints in matrix `C`.

`DT = delaunayTriangulation(x,y,z)` creates a 3-D Delaunay triangulation from the point coordinates in the column vectors, `x`, `y`, and `z`.

`DT = delaunayTriangulation()` creates an empty Delaunay triangulation.

Input Arguments

P

Input points, specified as a matrix whose columns are the *x*, *y*, (and possibly *z*) coordinates of the triangulation points. The row numbers of `P` are the vertex IDs in the triangulation.

x

x-coordinates vector, specified as a column vector containing the *x*-coordinates of the triangulation points.

y

y-coordinates vector, specified as a column vector containing the *y*-coordinates of the triangulation points.

z

z-coordinates vector, specified as a column vector containing the *z*-coordinates of the triangulation points.

C

Vertex IDs of constrained edges, specified as a 2-column matrix. Each row of `C` corresponds to a constrained edge and contains two IDs:

- `C(j,1)` is the ID of the vertex at the start of an edge.
- `C(j,2)` is the ID of the vertex at end of the edge.

You can specify edge constraints for 2-D triangulations only.

Properties

Points

Points in the triangulation, represented as a matrix containing the following information:

- Each row in `DT.Points` contains the coordinates of a vertex.
- Each row number of `DT.Points` is a vertex ID.

ConnectivityList

Triangulation connectivity list, represented as a matrix. This matrix contains the following information:

- Each row represents a triangle or tetrahedron in the triangulation.
- Each row number of `DT.ConnectivityList` is a "Triangle or Tetrahedron ID" on page 1-1309.
- Each element is a vertex ID.

Constraints

Constrained edges, represented as a two-column matrix of vertex IDs. Each row of `DT.Constraints` corresponds to a constrained edge and contains two IDs:

- `DT.Constraints(j,1)` is the ID of the vertex at the start of an edge.
- `DT.Constraints(j,2)` is the ID of the vertex at end of the edge.

`DT.Constraints` is an empty matrix when the triangulation has no constrained edges.

Methods

convexHull	Convex hull
isInterior	Test if triangle is in interior of 2-D constrained Delaunay triangulation
nearestNeighbor	Vertex closest to specified location
pointLocation	Triangle or tetrahedron containing specified location
voronoiDiagram	Voronoi diagram

Inherited Methods

barycentricToCartesian	Converts point coordinates from barycentric to Cartesian
cartesianToBarycentric	Converts point coordinates from Cartesian to barycentric
circumcenter	Circumcenter of triangle or tetrahedron
edgeAttachments	Triangles or tetrahedra attached to specified edge
edges	Triangulation edges
faceNormal	Triangulation face normal
featureEdges	Triangulation sharp edges
freeBoundary	Triangulation facets referenced by only one triangle or tetrahedron
incenter	Incenter of triangle or tetrahedron
isConnected	Test if two vertices are connected by edge
neighbors	Neighbors to specified triangle or tetrahedron
size	Size of triangulation connectivity list
vertexAttachments	Triangles or tetrahedra attached to specified vertex
vertexNormal	Triangulation vertex normal

Definitions

Delaunay Triangulation

In a 2-D Delaunay triangulation, the circumcircle associated with each triangle does not contain any points in its interior. Similarly, a 3-D Delaunay triangulation does not have any points in the interior of the circumsphere associated with each tetrahedron. This definition extends to N-D, although `delaunayTriangulation` supports only 2-D and 3-D.

Vertex ID

A row number of the matrix, `DT.Points`. Use this ID to refer a specific vertex in the triangulation.

Triangle or Tetrahedron ID

A row number of the matrix, `DT.ConnectivityList`. Use this ID to refer a specific triangle or tetrahedron.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

2-D Delaunay Triangulation

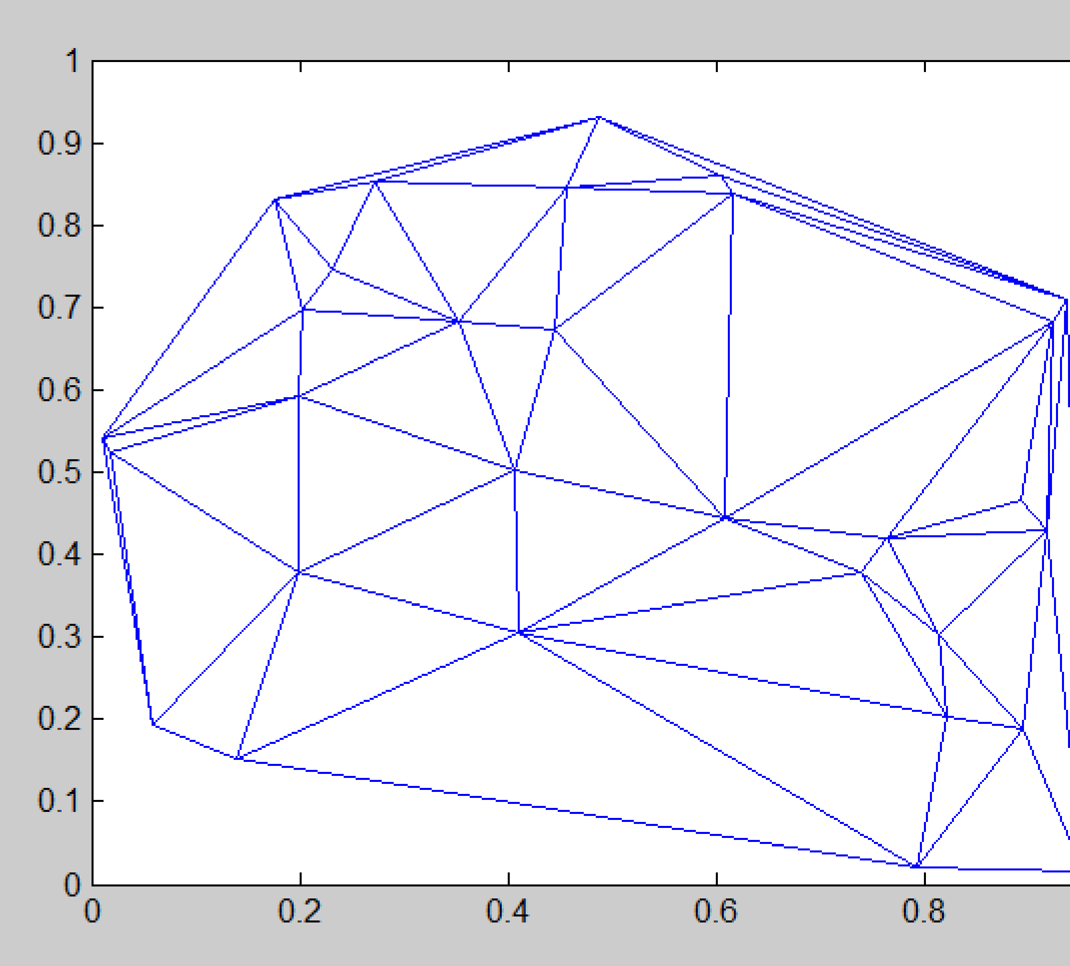
Create a 2-D `delaunayTriangulation` for 30 random points.

```
P = gallery('uniformdata',[30 2],0);
DT = delaunayTriangulation(P)
```

```
DT =

  delaunayTriangulation with properties:

              Points: [30x2 double]
    ConnectivityList: [50x3 double]
         Constraints: []
```

Plot the triangulation.

```
figure
triplot(DT)
```



3-D Delaunay Triangulation

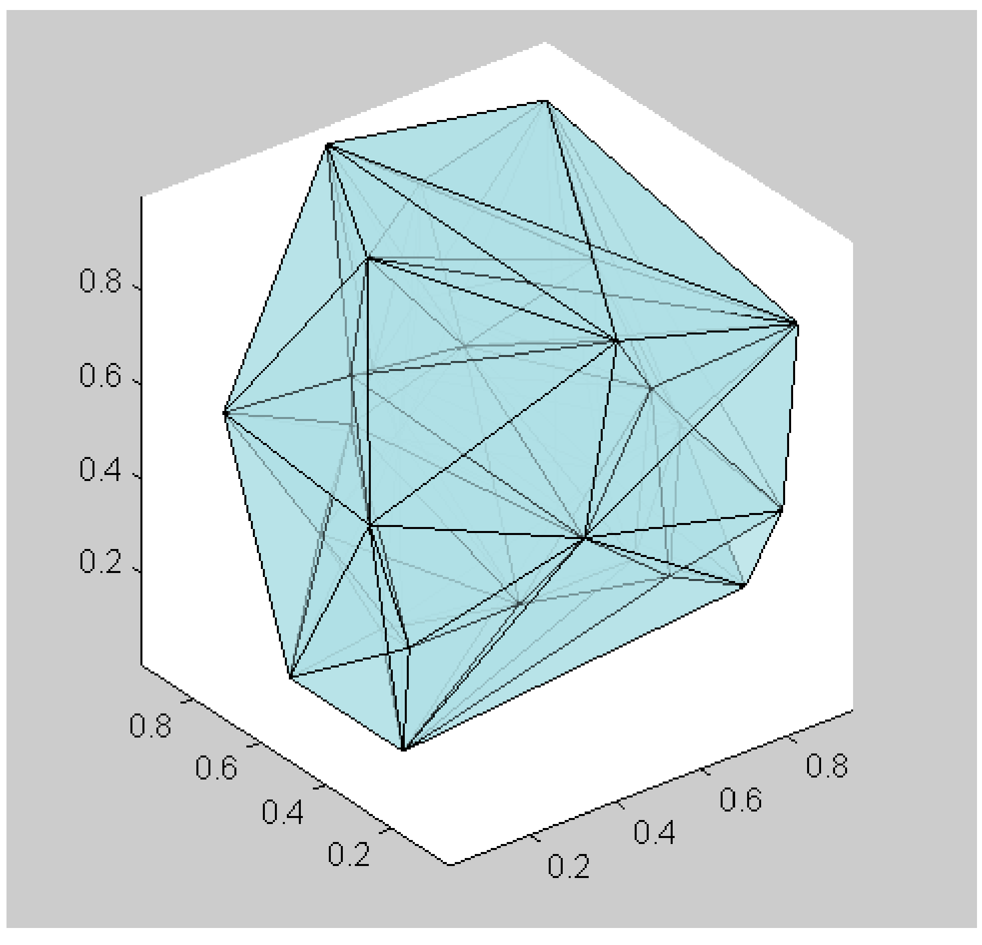
Create a 3-D `delaunayTriangulation` for 30 random points.

```
x = gallery('uniformdata',[30 1],0);
y = gallery('uniformdata',[30 1],1);
z = gallery('uniformdata',[30 1],2);
```

```
DT = delaunayTriangulation(x,y,z)

DT = 

  delaunayTriangulation with properties:

              Points: [30x3 double]
    ConnectivityList: [111x4 double]
         Constraints: []
```

Plot the triangulation at 30% opacity with a light blue face color.

```
faceColor  = [0.6875 0.8750 0.8984];
figure
tetramesh(DT,'FaceColor',faceColor,'FaceAlpha',0.3);
```



See Also

`triangulation` | `delaunay` | `delaunayn`

Purpose

Convex hull

Syntax

```
K = convexHull(DT)
[K,v] = convexHull(DT)
```

Description

`K = convexHull(DT)` returns the vertices of the convex hull.

`[K,v] = convexHull(DT)` also returns the area or volume bounded by the convex hull.

Input Arguments

DT

A Delaunay triangulation, see `delaunayTriangulation`.

Output Arguments

K

Convex hull vertices, returned as a matrix of vertex IDs. The shape of `K` depends on whether your triangulation is 2-D or 3-D:

- When `DT` is 2-D, `K` is a column vector containing the sequence of vertex IDs around the convex hull.
- When `DT` is 3-D, `K` is a triangulation connectivity list containing the triangles on the convex hull.

v

Area or volume bounded by the convex hull, returned as a scalar value.

Definitions

Vertex ID

A row number of the matrix, `DT.Points`. Use this ID to refer a specific vertex in the triangulation.

Examples

Convex Hull in 2-D Space

Create a Delaunay triangulation from a set of random points.

```
x = gallery('uniformdata',[10,1],0);
y = gallery('uniformdata',[10,1],1);
DT = delaunayTriangulation(x,y);
```

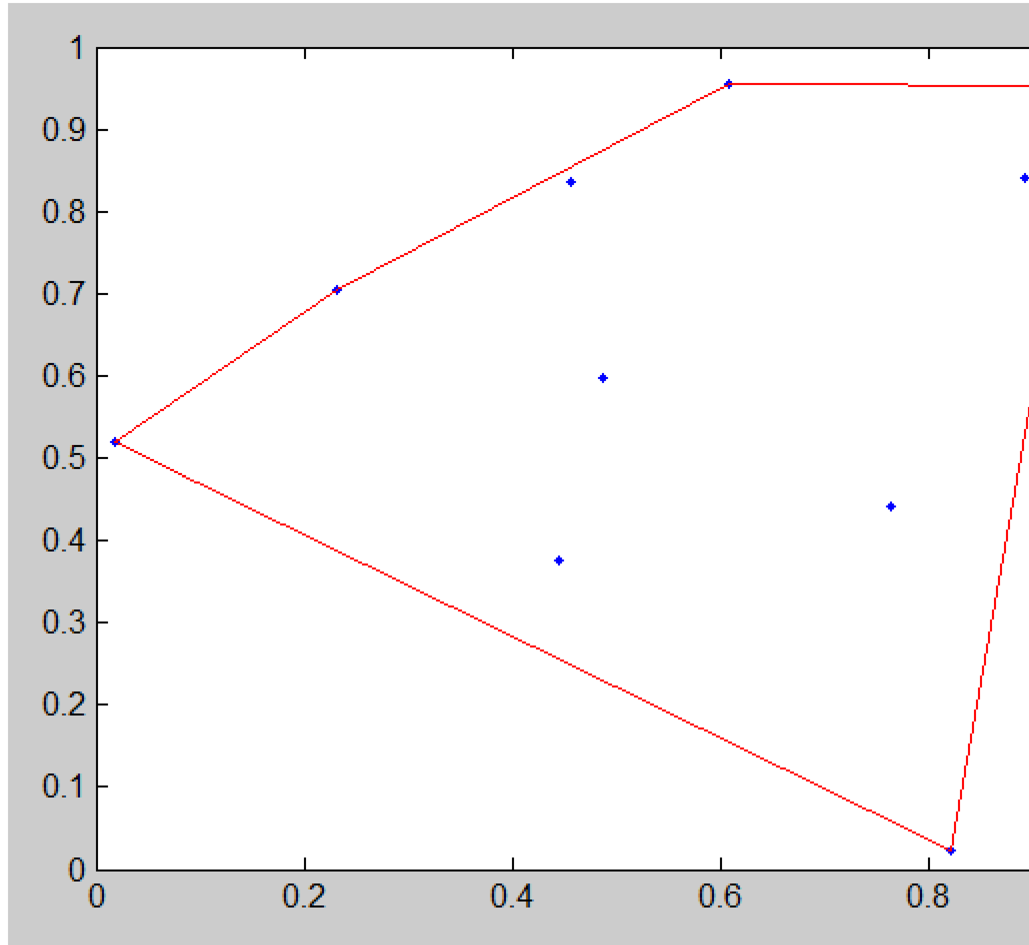
Calculate the convex hull.

```
k = convexHull(DT)
```

```
k =

     1
     3
     2
     8
     9
     1
```

Plot the points and highlight the convex hull in red.

```
plot(DT.Points(:,1),DT.Points(:,2), '.','markersize',10);
hold on;
plot(DT.Points(k,1),DT.Points(k,2),'r');
hold off;
```



Convex Hull in 3-D Space

Use `convexHull` to calculate the convex hull of a set of random points within a unit cube.

Create a Delaunay triangulation from a set of random points.

```
P = gallery('uniformdata',[25,3],1);
```

```
DT = delaunayTriangulation(P);
```

Calculate the convex hull and the volume bounded by the convex hull.

```
[K,v] = convexHull(DT);
```

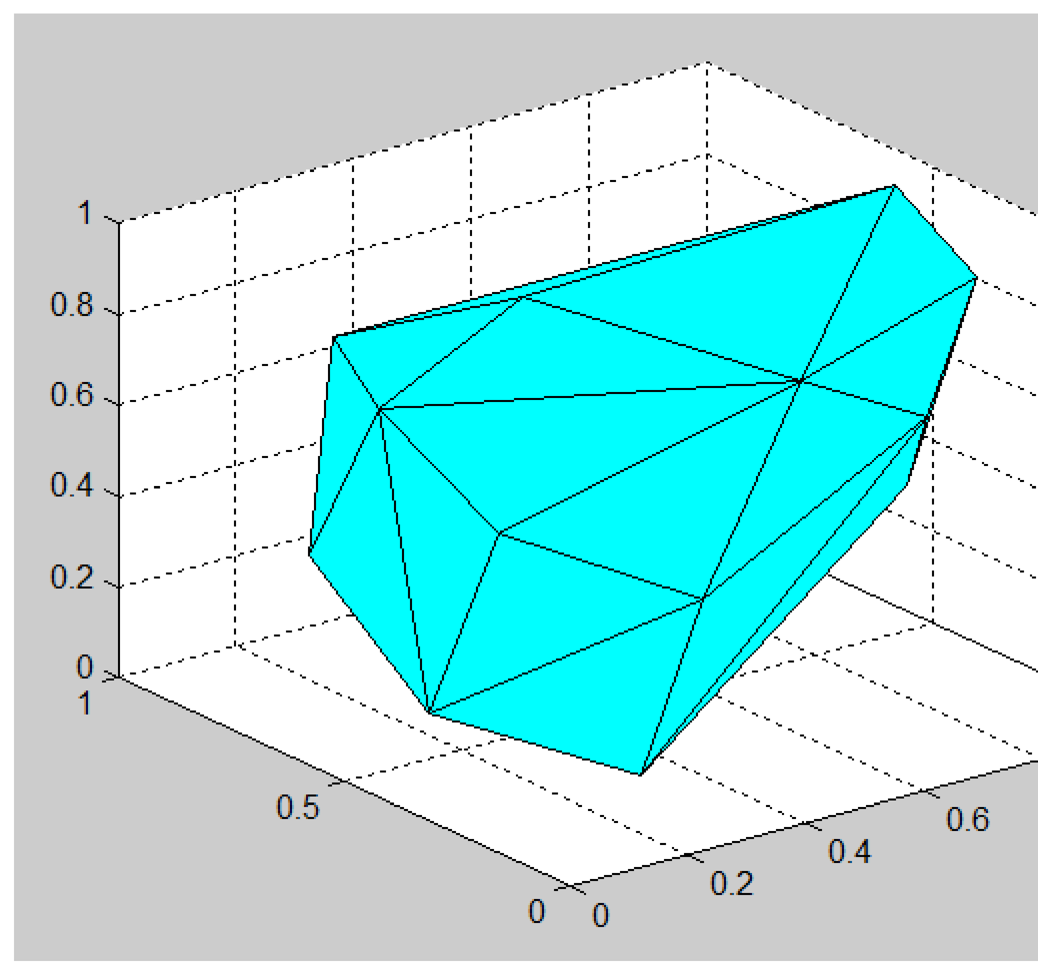
Examine the volume.

```
v
```

```
v =

    0.3561
```

Plot the convex hull.

```
trisurf(K,DT.Points(:,1),DT.Points(:,2),DT.Points(:,3),...
       'FaceColor','cyan')
```



See Also

`triangulation` | `convhulln` | `convhullvoronoiDiagram` |

delaunayTriangulation.isInterior

Purpose

Test if triangle is in interior of 2-D constrained Delaunay triangulation

Syntax

`tf = isInterior(DT)`

Description

`tf = isInterior(DT)` returns an array of logical values that indicate whether the triangles in a constrained Delaunay triangulation are inside the bounded geometric domain. A triangle, `DT.ConnectivityList(j,:)`, is classified as inside the domain when `tf(j)` is `true`. Otherwise, the triangle outside the domain.

Input Arguments

DT

A 2-D `delaunayTriangulation` that has a set of constrained edges that define a bounded geometric domain.

Output Arguments

tf

Logical values, returned as a column vector. Element `tf(j)` is `true` when the triangle whose ID is `j` is inside the domain of `DT`.

Definitions

Triangle ID

A row number of the matrix, `DT.ConnectivityList`. You use this ID to refer a specific triangle.

Examples

Find and Plot Triangles within a Boundary

Create a geometric domain whose shape is a square frame.

```
outerprofile = [-5 -5; -3 -5; -1 -5;  1 -5;
                 3 -5;  5 -5;  5 -3;  5 -1;
                 5  1;  5  3;  5  5;  3  5;
                 1  5; -1  5; -3  5; -5  5;
                -5  3; -5  1; -5 -1; -5 -3];

innerprofile = outerprofile.*0.5;
profile = [outerprofile; innerprofile];
```

Define the edge constraints.

```
outercons = [(1:19)' (2:20)'; 20 1;];
innercons = [(21:39)' (22:40)'; 40 21];
C = [outercons; innercons];
```

Create the constrained Delaunay triangulation.

```
DT = delaunayTriangulation(profile,C);
```

Plot the triangulation.

```
subplot(1,2,1);
triplot(DT);
```

Highlight the inner square in red.

```
hold on;
plot(DT.Points(innercons',1),DT.Points(innercons',2),...
     '-r','LineWidth',2);
```

Highlight the outer square in red and resize the *x* and *y* axes to make the plot square.

```
plot(DT.Points(outercons',1),DT.Points(outercons',2), ...
     '-r','LineWidth', 2);
axis equal;
```

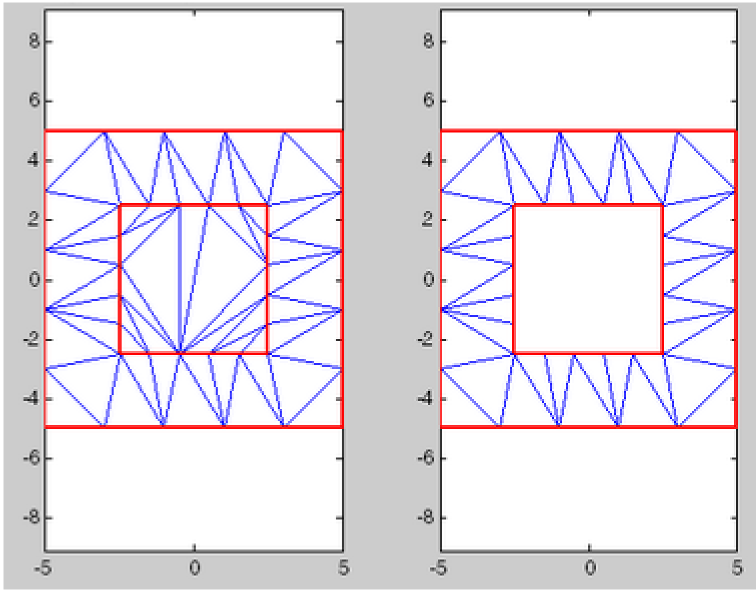
Plot only the triangles that lie inside of the domain.

```
hold off;
subplot(1,2,2);
inside = isInterior(DT);
triplot(DT.ConnectivityList(inside, :),DT.Points(:,1),DT.Points(:,2));
```

Highlight the inner and outer squares in red.

```
hold on;
plot(DT.Points(outercons',1),DT.Points(outercons',2), ...
```

```
     '-r','LineWidth', 2);
plot(DT.Points(innercons',1),DT.Points(innercons',2), ...
     '-r','LineWidth', 2);
axis equal;
hold off;
```



See Also `triangulation` |

Purpose

Vertex closest to specified location

Syntax

```
vi = nearestNeighbor(DT,QP)
vi = nearestNeighbor(DT,qx,qy)
vi = nearestNeighbor(DT,qx,qy,qz)
[vi,d] = nearestNeighbor( ___ )
```

Description

`vi = nearestNeighbor(DT,QP)` returns the IDs of the vertices closest to the query points in `QP`. Each row in matrix `QP` contains the *x*, *y*, (and possibly *z*) coordinates of a query point.

`vi = nearestNeighbor(DT,qx,qy)` specifies the *x* and *y* coordinates of the query points in 2-D as separate column vectors.

`vi = nearestNeighbor(DT,qx,qy,qz)` specifies the *x*, *y*, and *z* coordinates of the query points in 3-D as separate column vectors.

`[vi,d] = nearestNeighbor( ___ )` also returns the Euclidean distance between each query point and its nearest neighbor. You can specify both output arguments with any of the previous syntaxes.

`nearestNeighbor` does not support constrained Delaunay triangulations.

Input Arguments

DT

A Delaunay triangulation, see `delaunayTriangulation`.

QP

Query points, specified as a matrix whose columns are the *x*, *y*, (and possibly *z*) coordinates of the points to query.

qx

Query *x*-coordinates, specified as a column vector containing the *x*-coordinates of the points to query.

qy

Query *y*-coordinates, specified as a column vector containing the *y*-coordinates of the points to query.

qz

Query *z*-coordinates, specified as a column vector containing the *z*-coordinates of the points to query.

Output Arguments

vi

Vertex IDs of nearest neighbor, returned as a column vector. `vi(j)` is the "Vertex ID" on page 1-1322 of the nearest neighbor to the query point `QP(j)`.

d

Distance to nearest neighbor, returned as a column vector the same length as `vi`. Each value in `d` is the Euclidean distance between a query point and its nearest neighbor.

Definitions

Vertex ID

A row number of the matrix, `DT.Points`. Use this ID to refer a specific vertex in the triangulation.

Examples

Nearest Neighbors of Query Points in 2-D

Create a Delaunay triangulation from a set of random points.

```
x = gallery('uniformdata',[20 1],0);
y = gallery('uniformdata',[20 1],1);
DT = delaunayTriangulation(x,y);
```

Define two query points.

```
QP = [0.25 0.25; 0.5 0.5];
```

Find the nearest neighbors to the query points and the distances between them.

```
[vi,d] = nearestNeighbor(DT,QP);
```

Examine the vertex IDs of the nearest neighbors.

```
vi
```

```
vi =

    10
     4
```

Examine the coordinates of the nearest neighbors.

```
DT.Points(vi,:)
```

```
ans =

    0.4447    0.3759
    0.4860    0.5982
```

Examine the Euclidean distance between each query point and its nearest neighbor.

```
d
```

```
d =

    0.2319
    0.0992
```

See Also

triangulation | pointLocation |

delaunayTriangulation.pointLocation

Purpose

Triangle or tetrahedron containing specified location

Syntax

```
ti = pointLocation(DT,QP)
ti = pointLocation(DT,qx,qy)
ti = pointLocation(DT,qx,qy,qz)
[ti,B] = pointLocation( ___ )
```

Description

`ti = pointLocation(DT,QP)` returns the enclosing triangles or tetrahedra for each query point in `QP`. Each row in matrix `QP` contains the *x*, *y*, (and possibly *z*) coordinates of a query point.

`ti = pointLocation(DT,qx,qy)` specifies the *x* and *y* coordinates of the query points in 2-D as separate column vectors.

`ti = pointLocation(DT,qx,qy,qz)` specifies the *x*, *y*, *z* coordinates of the query points in 3-D as separate column vectors.

`[ti,B] = pointLocation( ___ )` also returns the barycentric coordinates of each query point with respect to its enclosing triangle or tetrahedron. You can specify both output arguments with any of the previous syntaxes.

Input Arguments

DT

A Delaunay triangulation, see `delaunayTriangulation`.

QP

Query points, specified as a matrix whose columns are the *x*, *y*, (and possibly *z*) coordinates of the points to query.

qx

Query *x*-coordinates, specified as a column vector containing the *x*-coordinates of the points to query.

qy

Query *y*-coordinates, specified as a column vector containing the *y*-coordinates of the points to query.

qz

Query *z*-coordinates, specified as a column vector containing the *z*-coordinates of the points to query.

Output Arguments

ti

IDs of enclosing triangles or tetrahedra, returned as a column vector. Each element in `ti` is the ID of a triangle or tetrahedron that encloses a query point. `ti(k)` encloses the `k`th query point.

`pointLocation` returns `NaN` values for any points outside the convex hull.

B

Barycentric coordinates, returned as matrix. The values at `B(j,:)` are the barycentric coordinates of the `j`th query point with respect to `ti(j)`.

Definitions

Triangle or Tetrahedron ID

A row number of the matrix, `DT.ConnectivityList`. Use this ID to refer a specific triangle or tetrahedron.

Examples

Find Enclosing Triangles in 2-D

Create a Delaunay triangulation from a set of random points.

```
x = gallery('uniformdata',[20 1],0);
y = gallery('uniformdata',[20 1],1);
DT = delaunayTriangulation(x,y);
```

Define two query points.

```
QP = [0.25 0.25; 0.5 0.5];
```

Find the IDs of the enclosing triangles for each query point.

```
ti = pointLocation(DT,QP)
```

```
ti =
```

```
     8
     7
```

Examine the connectivity of the enclosing triangles.

```
DT.ConnectivityList(ti,:)

ans =

    19    15     8
     4    10     6
```

Find Enclosing Tetrahera and Barycentric Coordinates in 3-D

Create a 3-D Delaunay triangulation from a set of random points.

```
x = gallery('uniformdata',[20 1],0);
y = gallery('uniformdata',[20 1],1);
z = gallery('uniformdata',[20 1],2);
DT = delaunayTriangulation(x,y,z);
```

Find the tetrahedra that enclose the query points and calculate the barycentric coordinates of the query points.

```
QP = [0.7 0.6 0.3; 0.5 0.5 0.5];
[ti,B] = pointLocation(DT,QP)

ti =

    57
    56

B =

    0.2796    0.0184    0.5286    0.1734
    0.3687    0.0149    0.5343    0.0821
```

See Also

`triangulation` | `nearestNeighbor` |

delaunayTriangulation.voronoiDiagram

Purpose

Voronoi diagram

Syntax

`[V,R] = voronoiDiagram(DT)`

Description

`[V,R] = voronoiDiagram(DT)` returns the Voronoi vertices, `V`, and the Voronoi regions, `R`, of the points, `DT.Points`.

The *Voronoi diagram* of a set of points, such as `DT.Points`, decomposes the space around each point, `DT.Points(j,:)`, into a region of influence, `R{j}`. Locations within the region, `R{j}`, are closer to point `j` than any other point in `DT.Points`. The region of influence is called the *Voronoi region*. The collection of all the Voronoi regions is the Voronoi diagram.

The Voronoi regions associated with points that lie on the convex hull of `DT.Points` are unbounded. Bounding edges of these regions radiate to infinity. The vertex at infinity is represented by the first vertex in `V`.

Input Arguments

DT

A Delaunay triangulation, see `delaunayTriangulation`.

Output Arguments

V

Voronoi vertices, returned as a matrix. Each row of `V` contains the coordinates of a Voronoi vertex.

R

Voronoi regions, returned as a vector cell array the same length as `DT.Points`. The elements of `R` are row numbers of `V`. The coordinates of the Voronoi vertices bounding a region are `V(R{j},:)`. The Voronoi region associated with the point `DT.Points(j)` is `R{j}`.

Examples

Compute the Voronoi Diagram of a 2-D Triangulation

Create a Delaunay triangulation from a set of points.

```
P = [ 0.5    0
      0      0.5
```

```
     -0.5   -0.5
     -0.2   -0.1
     -0.1    0.1
      0.1   -0.1
      0.1    0.1 ];
DT = delaunayTriangulation(P);
```

Calculate the Voronoi vertices and regions.

```
[V,R] = voronoiDiagram(DT);
```

Examine the connectivity of the Voronoi region associated with the third point in the triangulation.

```
R{3}

ans =

     1    10     7     4
```

Examine the coordinates of the Voronoi vertices bounding the region.

```
V(R{3},:)

ans =

       Inf       Inf
    0.7000   -1.6500
   -0.0500   -0.5250
   -1.7500    0.7500
```

The `Inf` values indicate that the region contains points on the convex hull.

See Also

`triangulation` | `convexHull` | `voronoi` | `voronoin`

Purpose

Remove files or graphics objects

Alternatives

As an alternative to the `delete` function, use the Current Folder browser.

Syntax

```
delete('fileName1', 'filename2', ...)
delete(h)
delete(handle_array)
delete fileName
```

Description

`delete('fileName1', 'filename2', ...)` deletes the files `fileName1`, `fileName2`, and so on, from the disk. `fileName` is a string and can be an absolute path or a path relative to the current folder. `fileName` also can include wildcards (*).

`delete(h)` deletes the graphics object with handle `h`. `h` can also be a vector or matrix of handles. Delete multiple objects by appending their handles as additional arguments, separated by commas. The function deletes objects without requesting verification, even if when they are windows.

`delete(handle_array)` is a method of the `handle` class. It removes from memory the handle objects referenced by `handle_array`.

When deleted, any references to the objects in `handle_array` become invalid. To remove the handle variables, use the `clear` function.

`delete fileName` is the command syntax. Delete multiple files by appending filenames, separated by spaces. When filenames contain space characters, you must use the functional form.

As `delete` does not ask for confirmation, to avoid accidentally losing files or graphics objects, make sure to specify accurately the items to delete. To move files to a different location when running `delete`, use the **General** preference for **Deleting files**, or the `recycle` function.

The `delete` function deletes files and graphics objects only. To delete folders, use `rmdir`.

Examples

Delete all files with a `.mat` extension in the `../mytests/` folder:

```
delete('../mytests/*.mat')
```

Create a figure and an axes, and then delete the axes:

```
hf = figure, ha = axes

hf =
     1
ha =
  170.0332

delete(ha)
```

The axes is deleted, but the figure remain. The axes handle `ha` remains in the workspace but no longer points to an object.

See Also

`dir` | `recycle` | `rmdir`

How To

- "Deleting Files and Folders Using Functions"
- "Path Names in MATLAB"

delete (COM)

Purpose

Remove COM control or server

Syntax

```
h.delete
delete(h)
```

Description

`h.delete` releases all interfaces derived from the specified COM server or control, and then deletes the server or control itself. This is different from releasing an interface, which releases and invalidates only that interface.

`delete(h)` is an alternate syntax.

Tips

COM functions are available on Microsoft Windows systems only.

See Also

`release` | `save (COM)` | `load (COM)` | `actxcontrol` | `actxserver`

Purpose

Remove file on FTP server

Syntax

```
delete(ftpobj,filename)
```

Description

`delete(ftpobj,filename)` removes the specified file from the current folder on the FTP server associated with `ftpobj`.

Input Arguments

ftpobj

FTP object created by `ftp`.

filename

String enclosed in single quotation marks that specifies the name of the file to delete.

Examples

Suppose that a hypothetical host, `ftp.testsite.com`, contains `myfile.m`. Connect to the server and delete the file:

```
test=ftp('ftp.testsite.com');
delete(test,'myfile.m');
```

See Also

`rmdir` | `ftp`

delete (handle)

Purpose Handle object destructor

Syntax

```
delete(h)
```

Description

`delete(h)` deletes a handle object, but does not clear the handle variable from the workspace. The handle variable is not valid once the handle object has been deleted.

A subclass of `handle` can implement a method named `delete` to perform cleanup tasks just before MATLAB destroys the handle object. MATLAB calls the `delete` method of any handle object (if it exists) when the object is destroyed. `h` is a scalar handle object.

A `delete` method should not generate errors or create new handles to the object being destroyed. If the `delete` method has a different signature (having output arguments or more than one input argument) it is not called when the handle objects is destroyed. See "Handle Class Destructor" information on defining destructors for `handle` subclasses.

See Also

`handle` | `(handle) isvalid`

Purpose

Remove serial port object from memory

Syntax

```
delete(obj)
```

Description

`delete(obj)` removes `obj` from memory, where `obj` is a serial port object or an array of serial port objects.

Tips

When you delete `obj`, it becomes an *invalid* object. Because you cannot connect an invalid serial port object to the device, you should remove it from the workspace with the `clear` command. If multiple references to `obj` exist in the workspace, then deleting one reference invalidates the remaining references.

If `obj` is connected to the device, it has a `Status` property value of `open`. If you issue `delete` while `obj` is connected, then the connection is automatically broken. You can also disconnect `obj` from the device with the `fclose` function.

If you use the `help` command to display help for `delete`, then you need to supply the pathname shown below.

```
help serial/delete
```

Examples

This example creates the serial port object `s` on a Windows platform, connects `s` to the device, writes and reads text data, disconnects `s` from the device, removes `s` from memory using `delete`, and then removes `s` from the workspace using `clear`.

```
s = serial('COM1');
fopen(s)
fprintf(s,'*IDN?')
idn = fscanf(s);
fclose(s)
delete(s)
clear s
```

See Also

`clear` | `fclose` | `isvalid` | `Status`

delete (timer)

Purpose Remove timer object from memory

Syntax

```
delete(obj)
```

Description

`delete(obj)` removes the timer object, `obj`, from memory. If `obj` is an array of timer objects, `delete` removes all the objects from memory.

When you delete a timer object, it becomes invalid and cannot be reused. Use the `clear` command to remove invalid timer objects from the workspace.

If multiple references to a timer object exist in the workspace, deleting the timer object invalidates the remaining references. Use the `clear` command to remove the remaining references to the object from the workspace.

See Also

`clear` | `isvalid(timer)` | `timer`

Purpose

Remove custom property from COM object

Syntax

```
h.deleteproperty('propertyname')
deleteproperty(h,'propertyname')
```

Description

`h.deleteproperty('propertyname')` deletes the property specified in the string `propertyname` from the custom properties belonging to object or interface `h`.

`deleteproperty(h,'propertyname')` is an alternate syntax.

You can only delete properties created with the `addproperty` function.

COM functions are available on Microsoft Windows systems only.

Examples

Remove a custom property from an instance of the MATLAB sample control:

1. Create an instance of the control:

   ```
   f = figure('position', [100 200 200 200]);
   h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f);
   h.get
   ```

   MATLAB displays its properties:

   ```
        Label: 'Label'
       Radius: 20
   ```

2. Add a custom property named `Position` and assign a value:

   ```
   h.addproperty('Position');
   h.Position = [200 120];
   h.get
   ```

   MATLAB displays (in part):

   ```
          Label: 'Label'
         Radius: 20
   ```

```
    Position: [200 120]
```

3 Delete the custom property `Position`:

```
h.deleteproperty('Position');
h.get
```

MATLAB displays the original list of properties:

```
     Label: 'Label'
    Radius: 20
```

See Also

`addproperty` | `get (COM)` | `set (COM)` | `inspect`

Purpose

Remove `tsdata.event` objects from `timeseries` object

Syntax

```
ts = delevent(ts,event)
ts = delevent(ts,events)
ts = delevent(ts,event,n)
```

Description

`ts = delevent(ts,event)` removes the `tsdata.event` object from the `ts.events` property, where `event` is an event name string.

`ts = delevent(ts,events)` removes the `tsdata.event` object from the `ts.events` property, where `events` is a cell array of event name strings.

`ts = delevent(ts,event,n)` removes the `nth` `tsdata.event` object from the `ts.events` property. `event` is the name of the `tsdata.event` object.

Examples

The following example shows how to remove an event from a `timeseries` object:

1. Create a time series.

   ```
   ts = timeseries(rand(5,4))
   ```

2. Create an event object called `'test'` such that the event occurs at time `3`.

   ```
   e = tsdata.event('test',3)
   ```

3. Add the event object to the time series `ts`.

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

4. Remove the event object from the time series `ts`.

   ```
   ts = delevent(ts,'test')
   ```

See Also

`addevent` | `timeseries` | `tsdata.event`

delsamplefromcollection

Purpose

Remove sample from `tscollection` object

Syntax

```
tsc = delsamplefromcollection(tsc,'Index',N)
tsc = delsamplefromcollection(tsc,'Value',Time)
```

Description

`tsc = delsamplefromcollection(tsc,'Index',N)` deletes samples from the `tscollection` object tsc. N specifies the indices of the tsc time vector that correspond to the samples you want to delete.

`tsc = delsamplefromcollection(tsc,'Value',Time)` deletes samples from the `tscollection` object tsc. Time specifies the time values that correspond to the samples you want to delete.

See Also

`addsampletocollection` | `tscollection`

Purpose

Access product examples in Help browser

Syntax

```
demo
demo type
demo type name
```

Description

`demo` displays the list of MATLAB examples in the Help browser.

`demo type` lists the examples for the specified product. Valid values for `type` are `matlab` or `simulink`.

`demo type name` lists the examples for products other than MATLAB or Simulink. Valid values for `type` include `matlab`, `simulink`, `toolbox`, or `blockset`.

Input Arguments

type - Product name or type

`matlab` (default) | `simulink` | `toolbox` | `blockset`

Product name or type, specified as one of these strings: `matlab`, `simulink`, `toolbox`, or `blockset`. For products other than MATLAB or Simulink, you must also specify a `name` input that corresponds to the product name.

name - Product name other than MATLAB or Simulink

string

Product name other than MATLAB or Simulink, specified as a string. If `name` requires multiple words, enclose it in single quotes.

On non-English systems, `name` sometimes uses the localized language.

Examples

MATLAB Examples

```
demo matlab
```

Statistics Toolbox™ Examples

```
demo toolbox statistics
```

Communications System Toolbox™ Examples

```
demo toolbox 'communications system'
```

Simulink Control Design™ Examples

```
demo simulink 'simulink control design'
```

Tips

- To access third-party and custom examples without using the `demo` command, open the Help browser and navigate to the documentation home page. Then, at the bottom of the page, click **Supplemental Software**.

See Also

`echodemo` | `grabcode` | `help` | `doc`

Purpose

List dependent folders for function or P-file

Syntax

```
list = depdir('file_name')
[list, prob_files, prob_sym,
   prob_strings] = depdir('file_name')
[...] = depdir('file_name1', 'file_name2',...)
```

Description

The `depdir` function lists the folders of all the functions that a specified function or P-file needs to operate. This function is useful for finding all the folders that need to be included with a run-time application and for determining the run-time path.

`list = depdir('file_name')` creates a cell array of strings containing the folders of all the function and P-files that `file_name.m` or `file_name.p` uses. This includes the second-level files that are called directly by `file_name`, as well as the third-level files that are called by the second-level files, and so on.

`[list, prob_files, prob_sym, prob_strings] = depdir('file_name')` creates three additional cell arrays containing information about any problems with the `depdir` search. `prob_files` contains filenames that `depdir` was unable to parse. `prob_sym` contains symbols that `depdir` was unable to find. `prob_strings` contains callback strings that `depdir` was unable to parse.

`[...]  = depdir('file_name1', 'file_name2',...)` performs the same operation for multiple files. The dependent folders of all files are listed together in the output cell arrays.

Examples

```
list = depdir('mesh')
```

See Also

`depfun`

depfun

Purpose

List dependencies of function or P-file

Syntax

```
list = depfun(fun)
[list,builtins] = depfun(fun)
[list,builtins,classes] = depfun(fun)
[list,builtins,classes,prob_files] = depfun(fun)
[list,builtins,classes,prob_files,~,
   eval_strings] = depfun(fun)
[list,builtins,classes,prob_files,~,eval_strings,
   called_from] = depfun(fun)
[list,builtins,classes,prob_files,~,eval_strings,called_from,
   opaque_classes] = depfun(fun)

___ = depfun(fun1,...,funN)

___ = depfun(___,option1,...,optionM)
```

Description

`list = depfun(fun)` returns the paths of MATLAB program files that `fun` requires to run, where `fun` is a function, a P-file, or a figure file for a user interface. The output list includes second-level functions that `fun` calls directly, functions that the second-level functions call, and so on. The `depfun` function also displays a report in the Command Window.

Note

- `depfun` does not always list all dependent files. For example, `depfun` does not list files hidden in callbacks or files whose names are constructed dynamically for evaluation.
- The output list often includes extra files that are not called when the function is actually evaluated.
- If `depfun` returns a parse error for any of the required functions, then the rest of the output of `depfun` might be incomplete. Correct the problematic files and invoke `depfun` again.

`[list,builtins] = depfun(fun)` returns the built-in functions that `fun` requires.

`[list,builtins,classes] = depfun(fun)` returns the MATLAB classes that `fun` requires.

`[list,builtins,classes,prob_files] = depfun(fun)` returns the files that `depfun` cannot parse, find, or access. If there are problematic files (that is, if `prob_files` is not empty), then the rest of the output of `depfun` might be incomplete. Correct the problematic files, and then invoke `depfun` again.

`[list,builtins,classes,prob_files,~,eval_strings] = depfun(fun)` returns a list of files that contain calls to `eval` or related functions (`evalin`, `feval`, or `evalc`). These calls potentially use functions that are not in `list`.

The fifth output argument for `depfun` is not implemented, and returns an empty structure array. To request arguments later in the list, use the tilde symbol (~) as a placeholder. The tilde suppresses the creation of an extra, empty variable.

`[list,builtins,classes,prob_files,~,eval_strings,called_from] = depfun(fun)` returns a cell array of indices that maps each function in `list` to the set of functions that call it. That is, `list{called_from{k}}` returns the paths to the functions that call the function in `list{k}`.

`[list,builtins,classes,prob_files,~,eval_strings,called_from,opaque_cl = depfun(fun)` returns the opaque classes that `fun` requires, such as Java or COM classes.

___ = depfun(fun1,...,funN) returns the functions required for multiple functions fun1,...,funN. You can request any of the outputs from the previous syntaxes.

___ = depfun(___,option1,...,optionM) modifies the output as described by the specified options. For example, specify '-toponly' to list only the functions that fun calls directly. You can precede the options with one or more function names.

Input Arguments

fun - Function name

string | cell array of strings

Function name, specified as a string or a cell array of strings.

fun must be on the MATLAB path, as determined by the which function. If the path contains any relative folders, then files in those folders also will have a relative path in the output.

Example: 'plot.m'

Example: 'plot.m','mesh.m'

Example: {'plot.m','mesh.m'}

Data Types
char | cell

option - Reporting option

'-toponly' | '-verbose' | '-quiet' | '-print'

Reporting option, specified as one of the following strings.

Option	Description
'-toponly'	Identify only the files used directly by the specified function(s), fun.
'-verbose'	Display additional internal messages.
'-quiet'	Display only error and warning messages, and not a summary report.

Option	Description
`'-print',filename`	Print a full report to the specified file.
`'-all'`	Display all outputs in the report, but return only the specified output arguments.
`'-expand'`	Include both indices and full paths in an exported report for the `call` or `called_from` list. Requires the `'-print'` option. This information does not appear in the Command Window report.
`'-calltree'`	Replaces the `called_from` list with a list of functions that each file calls, as derived from the `called_from` list.

Example: `'-print','myreport.txt'`

Data Types
`char`

Output Arguments

list - Paths to required files
cell array of strings

Paths to required files, returned as a cell array of strings.

builtins - Required MATLAB built-in functions
cell array of strings

Required MATLAB built-in functions, returned as a cell array of strings.

classes - Required MATLAB classes
cell array of strings

Required MATLAB classes, returned as a cell array of strings.

prob_files - Files that `depfun` cannot parse or access
structure

Files that `depfun` cannot parse or access, returned as a structure with these fields:

- `name` — Path to the file
- `listindex` — Index of the file in `list`
- `errmsg` — Error message that describes the problem
- `errid` — Error identifier, if present

eval_strings - Files that call evaluation functions
cell array of strings

Files that call evaluation functions, returned as a cell array of strings.

called_from - Indices to files that call each function in `list`
cell array

Indices to files that call each function in `list`, returned as a cell array. Each element of the cell array is a numeric array of indices that correspond to elements in `list`. The `list` and `called_from` outputs have the same length.

opaque_classes - Paths to opaque classes
cell array of strings

Paths to opaque classes, such as Java or COM classes, returned as a cell array of strings.

Examples

Identify Required Functions

Identify the MATLAB program files that `strtok.m` requires to run.

```
fun = 'strtok.m';
list = depfun(fun);
```

MATLAB displays a summary report in the Command Window and stores the list of files in variable `list`.

Identify Top-Level Dependencies Only

List only the program files that `strtok.m` calls directly.

```
fun = 'strtok.m';
list = depfun(fun,'-toponly')
```

```
==========================================================
depfun report summary:(top only)
----------------------------------------------------------
-> trace list:           2 files  (total)
                         1 files  (total arguments)
                         0 files  (arguments off MATLABPATH)
                         0 files  (argument duplicates on MATLABPATH)
----------------------------------------------------------
Notes: 1. Use argument  '-quiet' to not print this summary.
       2. Use arguments '-print','file' to produce a full
          report in file.
       3. Use argument  '-all' to display all possible
          left hand side arguments in the report(s).
==========================================================

list =

    'matlabroot\toolbox\matlab\strfun\strtok.m'
    'matlabroot\toolbox\matlab\strfun\@cell\strtok.m'
```

Determine Which File Invokes Each Function

One of the functions that `strtok.m` depends upon is `num2cell`. Determine which file in the dependency list calls `num2cell`.

Identify the functions that `strtok.m` requires (`list`) and the files that invoke each of those functions (`called_from`).

```
fun = 'strtok.m';
[list,builtins,classes,prob_files,~,eval_strings,called_from] = depfun
```

Find the index for `num2cell` within the dependency list.

```
num2cell_path = which('num2cell');
num2cell_index = find(ismember(list,num2cell_path))
```

```
num2cell_index =
     5
```

Identify the file that calls `num2cell`.

```
calls_num2cell = list{called_from{num2cell_index}}
```

```
calls_num2cell =
     matlabroot\toolbox\matlab\elmat\repmat.m
```

See Also

`depdir`

Concepts

- "Identify Program Dependencies"

Purpose

Matrix determinant

Syntax

```
d = det(X)
```

Description

`d = det(X)` returns the determinant of the square matrix `X`.

Tips

Testing singularity using `abs(det(X)) <= tolerance` is not recommended as it is difficult to choose the correct tolerance. The function `cond(X)` can check for singular and nearly singular matrices.

Algorithms

The determinant is computed from the triangular factors obtained by Gaussian elimination

```
[L,U] = lu(A)
s =  det(L)        % This is always +1 or -1
det(A) = s*prod(diag(U))
```

Examples

The statement `A = [1 2 3; 4 5 6; 7 8 9]`

produces

```
A =
     1     2     3
     4     5     6
     7     8     9
```

This happens to be a singular matrix, so `det(A)` produces a very small number. Changing `A(3,3)` with `A(3,3) = 0` turns `A` into a nonsingular matrix. Now `d = det(A)` produces `d = 27`.

See Also

`cond` | `condest` | `inv` | `lu` | `rref` | `mldivide` | `Arithmetic Operators \,/`

detrend

Purpose

Remove linear trends

Syntax

```
y = detrend(x)
y = detrend(x,'constant')
y = detrend(x,'linear',bp)
```

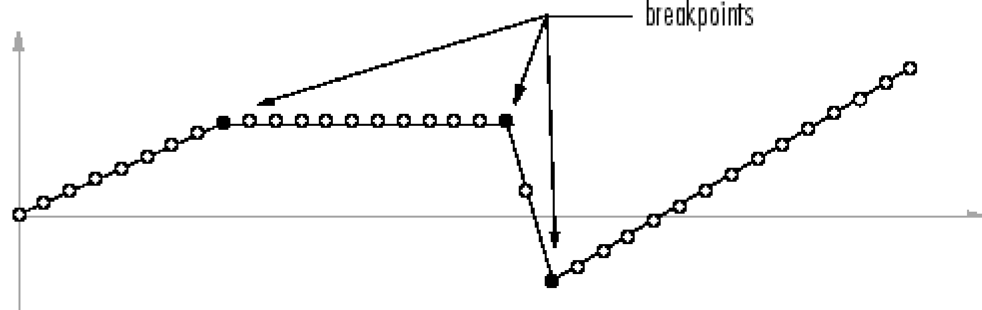
Description

`detrend` removes the mean value or linear trend from a vector or matrix, usually for FFT processing.

`y = detrend(x)` removes the best straight-line fit from vector `x` and returns it in `y`. If `x` is a matrix, `detrend` removes the trend from each column.

`y = detrend(x,'constant')` removes the mean value from vector `x` or, if `x` is a matrix, from each column of the matrix.

`y = detrend(x,'linear',bp)` removes a continuous, piecewise linear trend from vector `x` or, if `x` is a matrix, from each column of the matrix. Vector `bp` contains the indices of the breakpoints between adjacent linear segments. The breakpoint between two segments is defined as the data point that the two segments share.



`detrend(x,'linear')`, with no breakpoint vector specified, is the same as `detrend(x)`.

Examples

```
sig = [0 1 -2 1 0 1 -2 1 0];      % signal with no linear trend
trend = [0 1 2 3 4 3 2 1 0];      % two-segment linear trend
```

```
x = sig+trend;                      % signal with added trend
y = detrend(x,'linear',5)           % breakpoint at 5th element

y =

   -0.0000
    1.0000
   -2.0000
    1.0000
    0.0000
    1.0000
   -2.0000
    1.0000
   -0.0000
```

Note that the breakpoint is specified to be the fifth element, which is the data point shared by the two segments.

Algorithms

`detrend` computes the least-squares fit of a straight line (or composite line for piecewise linear trends) to the data and subtracts the resulting function from the data. To obtain the equation of the straight-line fit, use `polyfit`.

See Also

`polyfit`

deval

Purpose

Evaluate solution of differential equation problem

Syntax

```
sxint = deval(sol,xint)
sxint = deval(xint,sol)
sxint = deval(sol,xint,idx)
sxint = deval(xint,sol,idx)
[sxint, spxint] = deval(...)
```

Description

`sxint = deval(sol,xint)` and `sxint = deval(xint,sol)` evaluate the solution of a differential equation problem. `sol` is a structure returned by one of these solvers:

- An initial value problem solver (`ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`, `ode15i`)
- A delay differential equations solver (`dde23`, `ddesd`, or `ddensd`),
- A boundary value problem solver (`bvp4c` or `bvp5c`).

`xint` is a point or a vector of points at which you want the solution. The elements of `xint` must be in the interval `[sol.x(1),sol.x(end)]`. For each `i`, `sxint(:,i)` is the solution at `xint(i)`.

`sxint = deval(sol,xint,idx)` and `sxint = deval(xint,sol,idx)` evaluate as above but return only the solution components with indices listed in the vector `idx`.

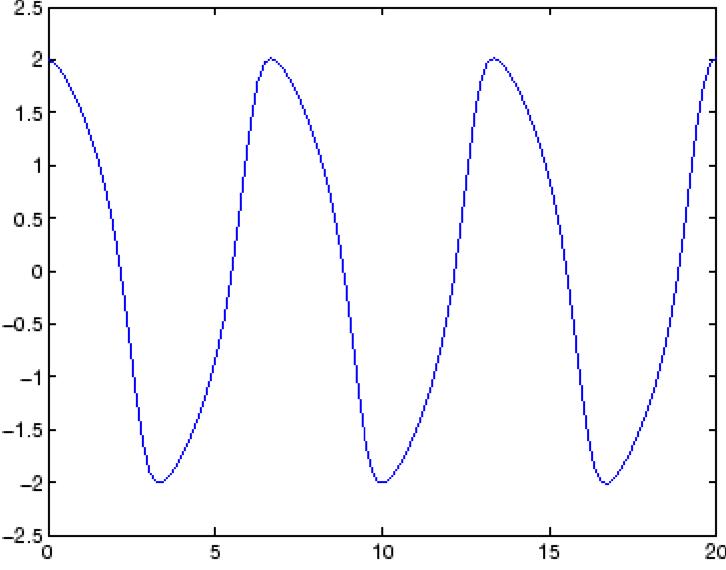
`[sxint, spxint] = deval(...)` also returns `spxint`, the value of the first derivative of the polynomial interpolating the solution.

Note For multipoint boundary value problems, the solution obtained by `bvp4c` or `bvp5c` might be discontinuous at the interfaces. For an interface point `xc`, `deval` returns the average of the limits from the left and right of `xc`. To get the limit values, set the `xint` argument of `deval` to be slightly smaller or slightly larger than `xc`.

Examples

This example solves the system $y' = \text{vdp1}(t,y)$ using `ode45`, and evaluates and plots the first component of the solution at 100 points in the interval `[0,20]`.

```
sol = ode45(@vdp1,[0 20],[2 0]);
x = linspace(0,20,100);
y = deval(sol,x,1);
plot(x,y);
```



See Also

`ode45` | `ode23` | `ode113` | `ode15s` | `ode23s` | `ode23t` | `ode23tb` | `ode15i` | `dde23` | `ddesd` | `ddensd` | `bvp4c` | `bvp5c`

diag

Purpose

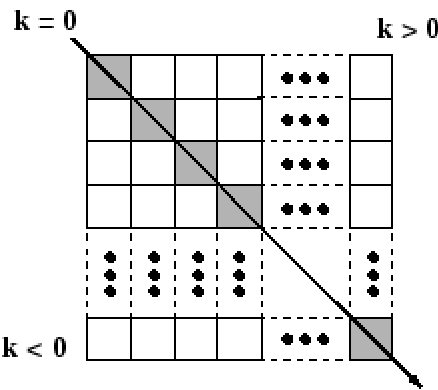
Diagonal matrices and diagonals of matrix

Syntax

```
X = diag(v,k)
X = diag(v)
v = diag(X,k)
v = diag(X)
```

Description

`X = diag(v,k)` when `v` is a vector of `n` components, returns a square matrix `X` of order `n+abs(k)`, with the elements of `v` on the `k`th diagonal. `k = 0` represents the main diagonal, `k > 0` above the main diagonal, and `k < 0` below the main diagonal.



`X = diag(v)` puts `v` on the main diagonal, same as above with `k = 0`.

`v = diag(X,k)` for matrix `X`, returns a column vector `v` formed from the elements of the `k`th diagonal of `X`.

`v = diag(X)` returns the main diagonal of `X`, same as above with `k = 0`.

Tips

`diag(diag(X))` is a diagonal matrix.

`sum(diag(X))` is the trace of `X`.

`diag([])` generates an empty matrix, (`[]`).

`diag(m-by-1,k)` generates a matrix of size `m+abs(k)`-by-`m+abs(k)`.

`diag(1-by-n,k)` generates a matrix of size `n+abs(k)`-by-`n+abs(k)`.

Examples

The statement

```
diag(-m:m)+diag(ones(2*m,1),1)+diag(ones(2*m,1),-1)
```

produces a tridiagonal matrix of order `2*m+1`.

See Also

`spdiags` | `tril` | `triu` | `blkdiag`

dialog

Purpose

Create and display empty dialog box

Syntax

```
h = dialog('PropertyName',PropertyValue,...)
```

Description

`h = dialog('PropertyName',PropertyValue,...)` returns a handle to a dialog box. The dialog box is a figure graphics object with properties recommended for dialog boxes. You can specify any valid figure property value except `DockControls`, which is always `off`.

The properties that `dialog` sets and their values are

Property	Value
'ButtonDownFcn'	'if isempty(allchild(gcbf)), close(gcbf), end'
'Colormap'	[]
'Color'	DefaultUicontrolBackgroundColor
'DockControls'	'off'
'HandleVisibility'	'callback'
'IntegerHandle'	'off'
'InvertHardcopy'	'off'
'MenuBar'	'none'
'NumberTitle'	'off'
'PaperPositionMode'	'auto'
'Resize'	'off'
'Visible'	'on'
'WindowStyle'	'modal'

Note By default, the dialog box is modal. A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

The default `ButtonDownFcn`, `if isempty(allchild(gcbf)), close(gcbf), end`, causes the dialog to terminate itself when clicked as long as it contains no child objects. Replace it with another callback or an empty callback if you do not want this behavior. You can do this only from a script or function if the dialog is modal (the default `WindowStyle`).

Any property you can specify for the `figure` function is valid for this function.

Examples

Create a default modal dialog box:

```
out = dialog;
```

Create a nonmodal dialog box with a name on its title bar:

```
out = dialog('WindowStyle', 'normal', 'Name', 'My Dialog');
```

See Also

`errordlg` | `helpdlg` | `inputdlg` | `listdlg` | `msgbox` | `questdlg` | `warndlg` | `figure` | `uiwait` | `uiresume`

diary

Purpose

Save Command Window text to file

Syntax

```
diary
diary('filename')
diary off
diary on
diary filename
```

Description

The `diary` function creates a log of keyboard input and the resulting text output, with some exceptions (see "Tips" on page 1-1360 for details). The output of `diary` is an ASCII file, suitable for searching in, printing, inclusion in most reports and other documents. If you do not specify `filename`, the MATLAB software creates a file named `diary` in the current folder.

`diary` toggles `diary` mode on and off. To see the status of `diary`, type `get(0,'Diary')`. MATLAB returns either `on` or `off` indicating the `diary` status.

`diary('filename')` writes a copy of all subsequent keyboard input and the resulting output (except it does not include graphics) to the named file, where `filename` is the full pathname or `filename` is in the current MATLAB folder. If the file already exists, output is appended to the end of the file. You cannot use a `filename` called `off` or `on`. To see the name of the `diary` file, use `get(0,'DiaryFile')`.

`diary off` suspends the diary.

`diary on` resumes diary mode using the current filename, or the default filename `diary` if none has yet been specified.

`diary filename` is the unquoted form of the syntax.

Tips

Because the output of `diary` is plain text, the file does not exactly mirror input and output from the Command Window:

- Output does not include graphics (figure windows).
- Syntax highlighting and font preferences are not preserved.

- Hidden components of Command Window output such as hyperlink information generated with `matlab:` are shown in plain text. For example, if you enter the following statement

  ```
  str = sprintf('%s%s', ...
     '<a href="matlab:magic(4)">', ...
     'Generate magic square</a>');
  disp(str)
  ```

  MATLAB displays

  Generate magic square

  However, the diary file, when viewed in a text editor, shows

  ```
  str = sprintf('%s%s', ...
     '<a href="matlab:magic(4)">', ...
     'Generate magic square</a>');
  disp(str)
  <a href="matlab:magic(4)">Generate magic square</a>
  ```

  If you view the output of diary in the Command Window, the Command Window interprets the `<a href ...>` statement and displays it as a hyperlink.

- Viewing the output of `diary` in a console window might produce different results compared to viewing `diary` output in the desktop Command Window. One example is using the `\r` option for the `fprintf` function; using the `\n` option might alleviate that problem.

See Also

`evalc`

How To

- "Command History"

Purpose

Differences and approximate derivatives

Syntax

```
Y = diff(X)
Y = diff(X,n)
Y = diff(X,n,dim)
```

Description

`Y = diff(X)` calculates differences between adjacent elements of `X`.

If `X` is a vector, then `diff(X)` returns a vector, one element shorter than `X`, of differences between adjacent elements:

```
[X(2)-X(1) X(3)-X(2) ... X(n)-X(n-1)]
```

If `X` is a matrix, then `diff(X)` returns a matrix of row differences:

```
[X(2:m,:)-X(1:m-1,:)]
```

In general, `diff(X)` returns the differences calculated along the first non-singleton (`size(X,dim) > 1`) dimension of `X`.

`Y = diff(X,n)` applies `diff` recursively `n` times, resulting in the nth difference. Thus, `diff(X,2)` is the same as `diff(diff(X))`.

`Y = diff(X,n,dim)` is the nth difference function calculated along the dimension specified by scalar `dim`. If order `n` equals or exceeds the length of dimension `dim`, `diff` returns an empty array.

Tips

Since each iteration of `diff` reduces the length of `X` along dimension `dim`, it is possible to specify an order `n` sufficiently high to reduce `dim` to a singleton (`size(X,dim) = 1`) dimension. When this happens, `diff` continues calculating along the next nonsingleton dimension.

Examples

The quantity `diff(y)./diff(x)` is an approximate derivative.

```
x = [1 2 3 4 5];
y = diff(x)
y =
     1     1     1     1
```

```
z = diff(x,2)
z =
     0     0     0
```

Given,

```
A = rand(1,3,2,4);
```

`diff(A)` is the first-order difference along dimension 2.

`diff(A,3,4)` is the third-order difference along dimension 4.

See Also

`gradient` | `prod` | `cumsum` | `sum`

diffuse

Purpose

Calculate diffuse reflectance

Syntax

`R = diffuse(Nx,Ny,Nz,S)`

Description

`R = diffuse(Nx,Ny,Nz,S)` returns the reflectance of a surface with normal vector components `[Nx,Ny,Nz]`. `S` specifies the direction to the light source. You can specify these directions as three vectors`[x,y,z]` or two vectors `[Theta Phi` (in spherical coordinates).

Lambert's Law: `R = cos(PSI)` where `PSI` is the angle between the surface normal and light source.

See Also

`specular` | `surfnorm` | `surfl`

How To

- "Lighting Overview"

Purpose

List folder contents

Syntax

```
dir
dir name
listing = dir(name)
```

Description

`dir` lists the files and folders in the MATLAB current folder. Results appear in the order returned by the operating system.

`dir name` lists the files and folders that match the string `name`. When name is a folder, `dir` lists the contents of the folder. Specify `name` using absolute or relative path names. You can use wildcards (*).

`listing = dir(name)` returns attributes about `name`.

Tips

- To obtain a list of available drives on Microsoft Windows platforms:

  Use the DOS `net use` command in the Command Window:

  ```
  dos('net use')
  ```

  Or

  ```
  [s,r] = dos('net use')
  ```

  MATLAB returns the results to the character array `r`.

- Short DOS file name support

  The MATLAB `dir` function is consistent with the Microsoft Windows operating system `dir` command in that both support short file names generated by DOS.

- Structure Results for Nonexistent Files

  When you run `dir` with an output argument and the results include a nonexistent file or a file that `dir` cannot query for some other reason, then `dir` returns the following default values:

  ```
  date: ''
  bytes: []
  ```

```
  isdir: 0
  datenum: []
```

The most common occurrence is on UNIX[1] platforms when `dir` queries a file that is a symbolic link, which points to a nonexistent target. A nonexistent target is a target that was moved, removed, or renamed. For example, if `my_file` in `my_dir` is a symbolic link to another file that was deleted, then running

```
r = dir('my_dir')
```

includes this result for `my_file`:

```
r(n) =
  name: 'my_file'
  date: ''
  bytes: []
  isdir: 0
  datenum: []
```

where *n* is the index for `my_file`, found by searching `r` by the `name` field.

Input Arguments

name

A string value specifying a file or folder name.

Output Arguments

listing

Field Name	Description	Class
`name`	File or folder name	char array
`date`	Modification date timestamp	char array

1. UNIX is a registered trademark of The Open Group in the United States and other countries.

Field Name	Description	Class
`bytes`	Size of the file in bytes	double
`isdir`	`1` if name is a folder; `0` if not	logical
`datenum`	Modification date as serial date number. The value is locale-dependent.	double

Examples

View the Contents of a Folder

View the contents of the `matlab/audiovideo` folder:

```
dir(fullfile(matlabroot, 'toolbox/matlab/audiovideo'))
```

Find Information in the Return Structure

Return the folder listing, restricted to files with a `.m` extension, to the variable `av_files`:

```
av_files = dir(fullfile(matlabroot, ...
               'toolbox/matlab/audiovideo/*.m'))
```

MATLAB returns the information in a structure array:

```
av_files =
25x1 struct array with fields:
    name
    date
    bytes
    isdir
    datenum
```

Index into the structure to access a particular item:

```
av_files(3).name
ans =
   audioplayerreg.m
```

– Use the Wildcard Character to Find Multiple Files

View the MAT-files in the current folder that include the term `my_data` by using the wildcard character:

```
dir *my_data*.mat
```

MATLAB returns all file names that match this specification. For instance, it returns the following if they are in the current folder:

```
old_my_data.mat  my_data_final.mat  my_data_test.mat
```

– Exclude Certain Files from the Output

Return the list of files in the current folder, excluding those files that `dir` cannot query:

```
 y = dir;
 y = y(find(~cellfun(@isempty,{y(:).date})));
```

– Find the Date a File Was Modified

To get the serial date number for the date and time a file was last modified, use the `datenum` field of the structure returned by the `dir` command:

```
DirInfo = dir('startup.m');
filedate = DirInfo.datenum
```

Using the `datenum` function to convert the string returned in the `date` field of the structure is not recommended as this can behave differently in different locales.

```
filedate = datenum(DirInfo.date)
```

Alternatives

Use the Current Folder browser to view the list of files in a folder.

See Also

`cd` | `fileattrib` | `isdir` | `ls` | `mkdir` | `rmdir` | `what`

How To

- “Working with Files and Folders”

- “Path Names in MATLAB”

FTP.dir

Purpose

View contents of folder on FTP server

Syntax

```
dir(ftpobj)
dir(ftpobj,folder)
details = dir(ftpobj,folder)
```

Description

`dir(ftpobj)` lists the files in current folder on the FTP server associated with `ftpobj`.

`dir(ftpobj,folder)` lists the files in the specified folder.

`details = dir(ftpobj,folder)` returns the results in a structure array that contains the name, modification date, and size of each file.

Input Arguments

ftpobj

FTP object created by `ftp`.

folder

String enclosed in single quotation marks that specifies the target folder. To specify the folder above the current one, use `'..'`.

Output Arguments

details

`m`-by-1 structure array, where `m` is the number of files in the folder. Each element of the structure array contains the following fields:

Field Name	Description	Data Type
`name`	File name	`char`
`date`	Modification date timestamp	`char`
`bytes`	Number of bytes allocated to the file	`double`

Field Name	Description	Data Type
isdir	If a folder, isdir = 1, otherwise isdir = 0	logical
datenum	Modification date as serial date number	char

Examples

Connect to the MathWorks FTP server and view the contents:

```
mw=ftp('ftp.mathworks.com');
dir(mw)
```

This code returns:

```
README    incoming  matlab    outgoing  pub       pubs
```

Continuing the previous example, save the folder contents to the structure m, close the connection, and view details about the pub subfolder:

```
m=dir(mw);
close(mw);

m(5)
```

This code returns:

```
ans =
       name: 'pub'
       date: '13-Aug-2008 00:00:00'
      bytes: 512
      isdir: 1
    datenum: 733633
```

See Also

cd | ftp | mkdir | rmdir

disp

Purpose Display text or array

Syntax

```
disp(X)
```

Description

`disp(X)` displays the contents of `X` without printing the variable name. `disp` does not display empty variables.

Input Arguments

X - Variable to display

variable name

Variable to display, specified by the variable name.

Examples

Display Matrix with Column Labels

```
X = gallery('uniformdata',[5 3],0);
disp('     Corn      Oats      Hay')
disp(X)
```

```
     Corn      Oats      Hay
    0.9501    0.7621    0.6154
    0.2311    0.4565    0.7919
    0.6068    0.0185    0.9218
    0.4860    0.8214    0.7382
    0.8913    0.4447    0.1763
```

Display Hyperlink in Command Window

Include the full hypertext string on a single line as input to `disp`.

```
X = '<a href = "http://www.mathworks.com">MathWorks Web Site</a>';
disp(X)
```

```
MathWorks Web Site
```

The `disp` function generates a hyperlink in the Command Window. Click the link to display the MathWorks home page in a MATLAB Web browser.

Display Multiple Items on Same Line

Concatenate strings together using the [] operator. Convert any numeric values to characters using the num2str function.

```
name = 'Alice';   age = 12;
X = [name, ' will be ', num2str(age), ' this year.'];
```

Display the string.

```
disp(X)
```

```
Alice will be 12 this year.
```

You also can use sprintf to create a string. Terminate the command with a semicolon to prevent "str = " from being displayed. Then, use disp to display the string.

```
name = 'Alice';   age = 12;
X = sprintf('%s will be %d this year.', name, age);
disp(X)
```

```
Alice will be 12 this year.
```

Alternatively, use fprintf to create and display the string. Unlike the sprintf function, fprintf does not display the "str = " text. However, you need to end the string with the newline (\n) metacharacter to terminate its display properly.

```
name = 'Alice';   age = 12;
str = fprintf('%s will be %d this year.\n', name, age);
```

```
Alice will be 12 this year.
```

Tips

- The disp function accepts only one input. To display more than one array or string, you can use concatenation or the sprintf or fprintf functions as shown in the Example, “Display Multiple Items on Same Line” on page 1-1373.

See Also

`format` | `int2str` | `num2str` | `rats` | `sprintf` | `fprintf` | `colon (:)`

disp (memmapfile)

Purpose

Information about memmapfile object

Syntax

```
disp(obj)
```

Description

`disp(obj)` displays all properties and their values for `memmapfile` object `obj`.

The MATLAB software also displays this information when you construct a `memmapfile` object or set any of the object's property values, provided you do not terminate the command to do so with a semicolon.

Examples

Construct an object `m` of class `memmapfile`:

```
m = memmapfile('records.dat',                        ...
               'Offset', 2048,                       ...
               'Format', {                           ...
                  'int16'  [2 2] 'model';            ...
                  'uint32' [1 1] 'serialno';         ...
                  'single' [1 3] 'expenses'});
```

Use `disp` to display all the object's current properties:

```
disp(m)
    Filename: 'd:\matlab\records.dat'
    Writable: false
      Offset: 2048
      Format: {'int16' [2 2] 'model'
               'uint32' [1 1] 'serialno'
               'single' [1 3] 'expenses'}
      Repeat: Inf
        Data: 753x1 struct array with fields:
            model
         serialno
         expenses
```

See Also

`memmapfile` | `get (memmapfile)`

disp (MException)

Purpose

Display MException object

Syntax

```
disp(exception)
disp(exception.property)
```

Description

disp(exception) displays all properties (fields) of MException object exception.

disp(exception.property) displays the specified property of MException object exception.

Examples

Using the surf command without input arguments throws an exception. Use disp to display the identifier, message, stack, and cause properties of the MException object:

```
try
   surf
catch exception
   disp(exception)
end

 MException object with properties:

    identifier: 'MATLAB:narginchk:notEnoughInputs'
       message: 'Not enough input arguments.'
         cause: {}
         stack: [1x1 struct]
```

Display only the stack property:

```
disp(exception.stack)
    file: 'C:\Program Files\MATLAB\toolbox\matlab\graph3d\surf.m'
graph3d\surf.m'
    name: 'surf'
    line: 54
```

See Also

`try` | `catch` | `error` | `assert` | `MException` | `getReport(MException)` | `throw(MException)` | `rethrow(MException)` | `throwAsCaller(MException)` | `addCause(MException)` | `isequal(MException)` | `eq(MException)` | `ne(MException)` | `last(MException)`

disp (serial)

Purpose

Serial port object summary information

Syntax

```
obj
disp(obj)
```

Description

`obj` or `disp(obj)` displays summary information for `obj`, a serial port object or an array of serial port objects.

Tips

In addition to the syntax shown above, you can display summary information for `obj` by excluding the semicolon when:

- Creating a serial port object
- Configuring property values using the dot notation

Use the display summary to quickly view the communication settings, communication state information, and information associated with read and write operations.

Examples

The following commands display summary information for the serial port object `s.` on a Windows platform

```
s = serial('COM1')
s.BaudRate = 300
s
```

Purpose

Information about timer object

Syntax

```
disp(obj)
obj
```

Description

`disp(obj)` displays summary information for the timer object, `obj`.

If `obj` is an array of timer objects, `disp` outputs a table of summary information about the timer objects in the array.

`obj`, that is, typing the object name alone, does the same as `disp(obj)`

In addition to the syntax shown above, you can display summary information for `obj` by excluding the semicolon when

- Creating a timer object, using the `timer` function
- Configuring property values using the dot notation

Examples

The following commands display summary information for timer object `t`.

```
t = timer

Timer Object: timer-1

   Timer Settings
      ExecutionMode: singleShot
             Period: 1
           BusyMode: drop
            Running: off

   Callbacks
           TimerFcn: []
           ErrorFcn: []
           StartFcn: []
            StopFcn: []
```

This example shows the format of summary information displayed for an array of timer objects.

```
t2 = timer;
disp(timerfind)

Timer Object Array
Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:     Name:
   1       singleShot      1        ''            timer-1
   2       singleShot      1        ''            timer-2
```

See Also

`timer` | `get(timer)`

Purpose

Display text and numeric expressions

Syntax

```
display(X)
```

Description

`display(X)` prints the value of `X`. MATLAB implicitly calls `display` after any variable or expression that is not terminated by a semicolon.

To customize the display of objects, overload the `disp` function instead of the `display` function. `display` calls `disp`.

Examples

Display a Matrix

```
X = magic(3);
display(X)
```

```
X =

     8     1     6
     3     5     7
     4     9     2
```

Input Arguments

X - Input value

variable | expression

Input value, specified as a variable or expression.

See Also

`disp` | `ans` | `sprintf`

Concepts

- "Implement disp Or disp and display"
- "Overloading Functions for Your Class"
- "Relationship Between disp and display"

dither

Purpose

Convert image, increasing apparent color resolution by dithering

Syntax

```
X = dither(RGB, map)
X = dither(RGB, map, Qm, Qe)
BW = dither(I)
```

Description

`X = dither(RGB, map)` creates an indexed image approximation of the RGB image in the array `RGB` by dithering the colors in the colormap `map`. The colormap cannot have more than 65,536 colors.

`X = dither(RGB, map, Qm, Qe)` creates an indexed image from `RGB`, where `Qm` specifies the number of quantization bits to use along each color axis for the inverse color map, and `Qe` specifies the number of quantization bits to use for the color space error calculations. If `Qe < Qm`, dithering cannot be performed, and an undithered indexed image is returned in `X`. If you omit these parameters, `dither` uses the default values `Qm = 5`, `Qe = 8`.

`BW = dither(I)` converts the grayscale image in the matrix `I` to the binary (black and white) image `BW` by dithering.

Class Support

`RGB` can be `uint8`, `uint16`, `single`, or `double`. `I` can be `uint8`, `uint16`, `int16`, `single`, or `double`. All other input arguments must be `double`. `BW` is `logical`. `X` is `uint8`, if it is an indexed image with 256 or fewer colors; otherwise, it is `uint16`.

Algorithms

`dither` increases the apparent color resolution of an image by applying Floyd-Steinberg's error diffusion dither algorithm.

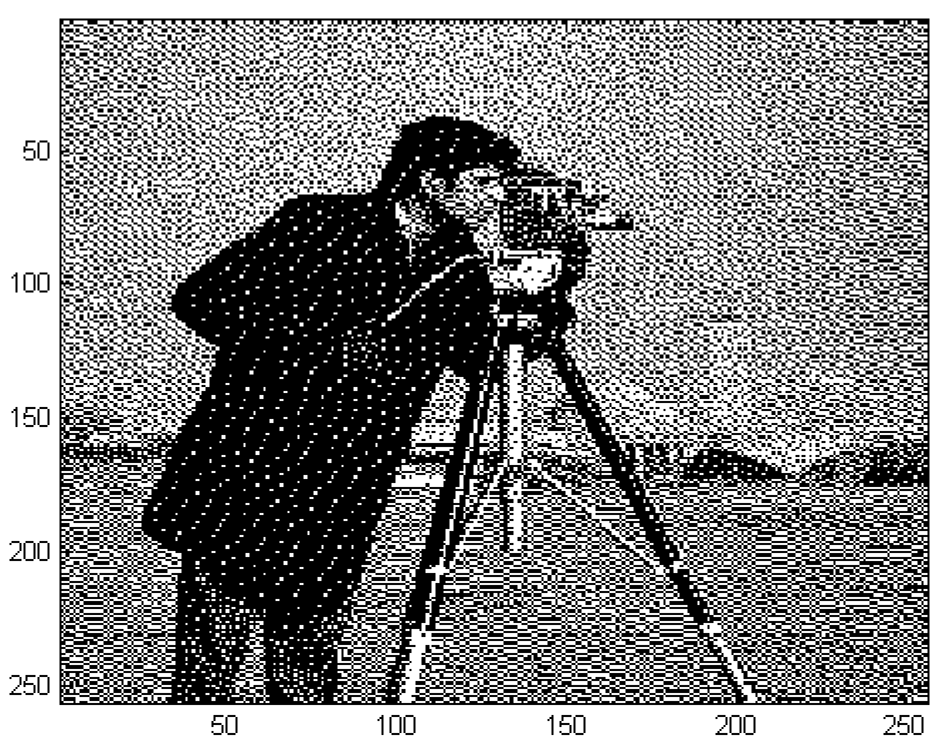
Examples

Read and display the `cameraman` image. Display the image as a gray scale image using `imagesc`.

```
I = imread('cameraman.tif');
figure; imagesc(I);
colormap(gray);
```

Apply dithering to get an indexed image as,

```
BW = dither(I);
figure; imagesc(BW);
colormap(gray);
```



References

[1] Floyd, R. W., and L. Steinberg, "An Adaptive Algorithm for Spatial Gray Scale," *International Symposium Digest of Technical Papers,* Society for Information Displays, 1975, p. 36.

[2] Lim, Jae S., *Two-Dimensional Signal and Image Processing*, Englewood Cliffs, NJ, Prentice Hall, 1990, pp. 469-476.

See Also

`rgb2ind`

Purpose

Compute divergence of vector field

Syntax

```
div = divergence(X,Y,Z,U,V,W)
div = divergence(U,V,W)
div = divergence(X,Y,U,V)
div = divergence(U,V)
```

Description

`div = divergence(X,Y,Z,U,V,W)` computes the divergence of a 3-D vector field `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`.

`div = divergence(U,V,W)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`div = divergence(X,Y,U,V)` computes the divergence of a 2-D vector field `U`, `V`.

The arrays `X` and `Y`, which define the coordinates for `U` and `V`, must be monotonic, but do not need to be uniformly spaced. `X` and `Y` must have the same number of elements, as if produced by `meshgrid`.

`div = divergence(U,V)` assumes `X` and `Y` are determined by the expression

```
[X Y] = meshgrid(1:n,1:m)
```

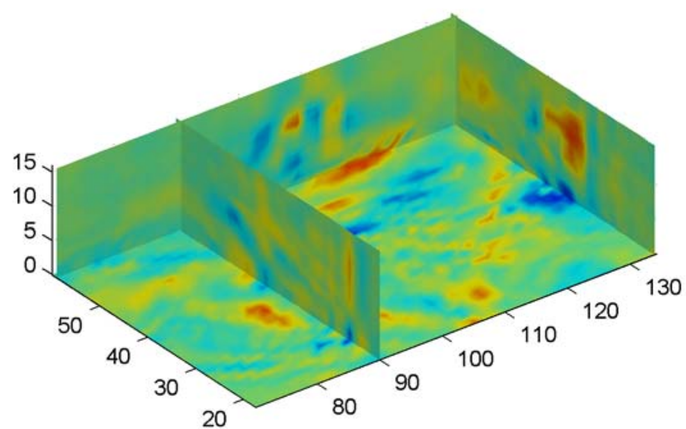
where `[m,n] = size(U)`.

Examples

This example displays the divergence of vector volume data as slice planes, using color to indicate divergence.

```
load wind
div = divergence(x,y,z,u,v,w);
```

```
h = slice(x,y,z,div,[90 134],59,0);
shading interp
daspect([1 1 1])
axis tight
camlight
set([h(1),h(2)],'ambientstrength',.6)
```



See Also

`streamtube` | `curl` | `isosurface`

How To

- "Displaying Divergence with Stream Tubes"

Purpose

Read ASCII-delimited file of numeric data into matrix

Syntax

```
M = dlmread(filename)
M = dlmread(filename, delimiter)
M = dlmread(filename, delimiter, R, C)
M = dlmread(filename, delimiter, range)
```

Description

`M = dlmread(filename)` reads the ASCII-delimited numeric data file `filename`, and returns the data in output matrix `M`. The `filename` input is a string enclosed in single quotes. `dlmread` infers the delimiter from the formatting of the file.

`M = dlmread(filename, delimiter)` reads data from the file, using the specified `delimiter`. Use `'\t'` to specify a tab delimiter.

`M = dlmread(filename, delimiter, R, C)` reads data whose upper left corner is at row `R` and column `C` in the file. Values `R` and `C` are zero-based, so that `R=0`, `C=0` specifies the first value in the file.

`M = dlmread(filename, delimiter, range)` reads the range specified by `range = [R1 C1 R2 C2]` where `(R1,C1)` is the upper left corner of the data to read and `(R2,C2)` is the lower right corner. You can also specify the range using spreadsheet notation, such as `range = 'A1..B7'`.

Tips

- All data in the input file must be numeric. `dlmread` does not read files that contain nonnumeric data, even if the specified rows and columns contain only numeric data.
- When `dlmread` infers the delimiter from the formatting of the file, it treats repeated white spaces as a single delimiter. By contrast, if you specify a delimiter, `dlmread` treats any repeated delimiter character as a separate delimiter.
- If you want to specify an `R`, `C`, or `range` input, but not a `delimiter`, set the `delimiter` argument to the empty string, (two consecutive single quotes with no spaces in between, `''`). For example,

  ```
  M = dlmread('myfile.dat', '', 5, 2)
  ```

In this case, `dlmread` treats repeated white spaces as a single delimiter.

- `dlmread` fills empty delimited fields with zero. If each line ends with a nonspace delimiter, such as a semicolon, the output matrix contains an additional last column of zeros.
- `dlmread` imports any complex number as a whole into a complex numeric field. Valid forms for a complex number are

`—<real>—<imag>i|j`	Example: `5.7-3.1i`
`—<imag>i|j`	Example: `-7j`

Embedded white-space in a complex number is invalid and is regarded as a field delimiter.

Examples

Example 1

Export a 5-by-8 test matrix `M` to a file, and read it with `dlmread`, first with no arguments other than the filename:

```
M = gallery('integerdata', 100, [5 8], 0);
dlmwrite('myfile.txt', M, 'delimiter', '\t')

dlmread('myfile.txt')
ans =
    96    77    62    41     6    21     2    42
    24    46    80    94    36    20    75    85
    61     2    93    92    82    61    45    53
    49    83    74    42     1    28    94    21
    90    45    18    90    14    20    47    68
```

Now read a portion of the matrix by specifying the row and column of the upper left corner:

```
dlmread('myfile.txt', '\t', 2, 3)
ans =
    92    82    61    45    53
```

```
    42     1    28    94    21
    90    14    20    47    68
```

This time, read a different part of the matrix using a range specifier:

```
dlmread('myfile.txt', '\t', 'C1..G4')
ans =
    62    41     6    21     2
    80    94    36    20    75
    93    92    82    61    45
    74    42     1    28    94
```

Example 2

Export matrix `M` to a file, and then append an additional matrix to the file that is offset one row below the first:

```
M = magic(3);
dlmwrite('myfile.txt', [M*5 M/5], ' ')

dlmwrite('myfile.txt', [M/3], '-append', ...
   'roffset', 1, 'delimiter', ' ')

type myfile.txt

40 5 30 1.6 0.2 1.2
15 25 35 0.6 1 1.4
20 45 10 0.8 1.8 0.4

2.6667 0.33333 2
1 1.6667 2.3333
1.3333 3 0.66667
```

When `dlmread` imports these two matrices from the file, it pads the smaller matrix with zeros:

```
dlmread('myfile.txt')
   40.0000    5.0000   30.0000    1.6000    0.2000    1.2000
```

```
   15.0000   25.0000   35.0000    0.6000    1.0000    1.4000
   20.0000   45.0000   10.0000    0.8000    1.8000    0.4000
    2.6667    0.3333    2.0000         0         0         0
    1.0000    1.6667    2.3333         0         0         0
    1.3333    3.0000    0.6667         0         0         0
```

See Also

dlmwrite | textscan

Purpose

Write matrix to ASCII-delimited file

Syntax

```
dlmwrite(filename, M)
dlmwrite(filename, M, 'D')
dlmwrite(filename, M, 'D', R, C)
dlmwrite(filename, M, 'attrib1', value1, 'attrib2', value2,
   ...)
dlmwrite(filename, M, '-append')
dlmwrite(filename, M, '-append', attribute-value list)
```

Description

`dlmwrite(filename, M)` writes matrix `M` into an ASCII format file using the default delimiter (,) to separate matrix elements. The data is written starting at the first column of the first row in the destination file, `filename`. The `filename` input is a string enclosed in single quotes.

`dlmwrite(filename, M, 'D')` writes matrix `M` into an ASCII format file, using delimiter `D` to separate matrix elements. The data is written starting at the first column of the first row in the destination file, `filename`. A comma (,) is the default delimiter. Use `\t` to produce tab-delimited files.

`dlmwrite(filename, M, 'D', R, C)` writes matrix `M` into an ASCII format file, using delimiter `D` to separate matrix elements. The data is written starting at row `R` and column `C` in the destination file, `filename`. `R` and `C` are zero based, so that `R=0`, `C=0` specifies the first value in the file, which is the upper left corner.

`dlmwrite(filename, M, 'attrib1', value1, 'attrib2', value2, ...)` is an alternate syntax to those shown above, in which you specify any number of attribute-value pairs in any order in the argument list. Each attribute must be immediately followed by a corresponding value (see the table below).

Attribute	Value
delimiter	Delimiter string to be used in separating matrix elements
newline	Character(s) to use in terminating each line (see table below)
roffset	Offset, in rows, from the top of the destination file to where matrix data is to be written. Offset is zero based.
coffset	Offset, in columns, from the left side of the destination file to where matrix data is to be written. Offset is zero based.
precision	Numeric precision to use in writing data to the file. Specify the number of significant digits or a C-style format string starting in `%`, such as `'%10.5f'`.

This table shows which values you can use when setting the **newline** attribute.

Line Terminator	Description
'pc'	PC terminator (implies carriage return/line feed (`CR`/`LF`))
'unix'	UNIX terminator (implies line feed (`LF`))

`dlmwrite(filename, M, '-append')` appends the matrix to the end of the file. If you do not specify `'-append'`, `dlmwrite` overwrites any existing data in the file.

`dlmwrite(filename, M, '-append', attribute-value list)` accepts a list of `attribute-value` pairs. If you specify `'-append'` and row or column offsets, `dlmwrite` calculates the offset from the end of the original contents of the file.

Tips

The resulting file is readable by spreadsheet programs. Alternatively, if your system has Excel for Windows installed, you can create a spreadsheet using `xlswrite`.

The `dlmwrite` function does not accept cell arrays for the input matrix `M`. To export a cell array that contains only numeric data, use `cell2mat` to convert the cell array to a numeric matrix before calling `dlmwrite`. For other cases, use low-level export functions. For more information, see "Exporting a Cell Array to a Text File" in the MATLAB Data Import and Export documentation.

Examples

Example 1

Export matrix `M` to a file delimited by the tab character and using a precision of six significant digits:

```
dlmwrite('myfile.txt', M, 'delimiter', '\t', ...
         'precision', 6)
type myfile.txt

0.893898      0.284409      0.582792      0.432907
0.199138      0.469224      0.423496      0.22595
0.298723      0.0647811     0.515512      0.579807
0.661443      0.988335      0.333951      0.760365
```

Example 2

Export matrix `M` to a file using a precision of six decimal places and the conventional line terminator for the PC platform:

```
dlmwrite('myfile.txt', m, 'precision', '%.6f', ...
         'newline', 'pc')
type myfile.txt

16.000000,2.000000,3.000000,13.000000
5.000000,11.000000,10.000000,8.000000
9.000000,7.000000,6.000000,12.000000
4.000000,14.000000,15.000000,1.000000
```

Example 3

Export matrix `M` to a file, and then append an additional matrix to the file that is offset one row below the first:

```
M = magic(3);
dlmwrite('myfile.txt', [M*5 M/5], ' ')

dlmwrite('myfile.txt', rand(3), '-append', ...
   'roffset', 1, 'delimiter', ' ')

type myfile.txt

40 5 30 1.6 0.2 1.2
15 25 35 0.6 1 1.4
20 45 10 0.8 1.8 0.4

0.81472 0.91338 0.2785
0.90579 0.63236 0.54688
0.12699 0.09754 0.95751
```

When `dlmread` imports these two matrices from the file, it pads the smaller matrix with zeros:

```
dlmread('myfile.txt')
   40.0000    5.0000   30.0000   1.6000   0.2000   1.2000
   15.0000   25.0000   35.0000   0.6000   1.0000   1.4000
   20.0000   45.0000   10.0000   0.8000   1.8000   0.4000
    0.8147    0.9134    0.2785   0        0        0
    0.9058    0.6324    0.5469   0        0        0
    0.1270    0.0975    0.9575   0        0        0
```

See Also

`dlmread` | `xlswrite`

Purpose

Dulmage-Mendelsohn decomposition

Syntax

```
p = dmperm(A)
[p,q,r,s,cc,rr] = dmperm(A)
```

Description

`p = dmperm(A)` finds a vector `p` such that `p(j) = i` if column `j` is matched to row `i`, or zero if column `j` is unmatched. If `A` is a square matrix with full structural rank, `p` is a maximum matching row permutation and `A(p,:)` has a zero-free diagonal. The structural rank of `A` is `sprank(A) = sum(p>0)`.

`[p,q,r,s,cc,rr] = dmperm(A)` where `A` need not be square or full structural rank, finds the Dulmage-Mendelsohn decomposition of `A`. `p` and `q` are row and column permutation vectors, respectively, such that `A(p,q)` has a block upper triangular form. `r` and `s` are index vectors indicating the block boundaries for the fine decomposition. `cc` and `rr` are vectors of length five indicating the block boundaries of the coarse decomposition.

`C = A(p,q)` is split into a 4-by-4 set of coarse blocks:

```
A11  A12  A13  A14
0      0  A23  A24
0      0    0  A34
0      0    0  A44
```

where `A12`, `A23`, and `A34` are square with zero-free diagonals. The columns of `A11` are the unmatched columns, and the rows of `A44` are the unmatched rows. Any of these blocks can be empty. In the coarse decomposition, the `(i,j)th` block is `C(rr(i):rr(i+1)-1,cc(j):cc(j+1)-1)`. For a linear system,

- `[A11 A12]` is the underdetermined part of the system—it is always rectangular and with more columns and rows, or `0`-by-`0`,
- `A23` is the well-determined part of the system—it is always square, and

- `[A34 ; A44]` is the overdetermined part of the system—it is always rectangular with more rows than columns, or `0`-by-`0`.

The structural rank of `A` is `sprank(A) = rr(4)-1`, which is an upper bound on the numerical rank of `A`. `sprank(A) = rank(full(sprand(A)))` with probability 1 in exact arithmetic.

The `A23` submatrix is further subdivided into block upper triangular form via the fine decomposition (the strongly connected components of `A23`). If `A` is square and structurally nonsingular, `A23` is the entire matrix.

`C(r(i):r(i+1)-1,s(j):s(j+1)-1)` is the `(i,j)`th block of the fine decomposition. The `(1,1)` block is the rectangular block `[A11 A12]`, unless this block is `0`-by-`0`. The `(b,b)` block is the rectangular block `[A34 ; A44]`, unless this block is `0`-by-`0`, where `b = length(r)-1`. All other blocks of the form `C(r(i):r(i+1)-1,s(i):s(i+1)-1)` are diagonal blocks of `A23`, and are square with a zero-free diagonal.

Tips

If `A` is a reducible matrix, the linear system $Ax=b$ can be solved by permuting `A` to a block upper triangular form, with irreducible diagonal blocks, and then performing block backsubstitution. Only the diagonal blocks of the permuted matrix need to be factored, saving fill and arithmetic in the blocks above the diagonal.

In graph theoretic terms, `dmperm` finds a maximum-size matching in the bipartite graph of `A`, and the diagonal blocks of `A(p,q)` correspond to the strong Hall components of that graph. The output of `dmperm` can also be used to find the connected or strongly connected components of an undirected or directed graph. For more information see Pothen and Fan [1].

References

[1] Pothen, Alex and Chin-Ju Fan “Computing the Block Triangular Form of a Sparse Matrix” *ACM Transactions on Mathematical Software* Vol 16, No. 4 Dec. 1990, pp. 303-324.

See Also

`sprank`

Purpose

Reference page in Help browser

Syntax

```
doc
doc name
```

Description

`doc` opens the Help browser. If the Help browser is already open, but not visible, then `doc` brings it to the foreground and opens a new tab.

`doc name` displays documentation for the functionality specified by `name`, such as a function, class, block, or product.

- If there is a MathWorks reference page or product page corresponding to `name`, then `doc` displays the page in the Help browser. The `doc` command does not display third-party or custom HTML documentation.
- If there is no reference or product page corresponding to `name`, then `doc` searches for help text in a file named *`name`*`.m`. When help text is available, `doc` displays it in a Web browser.
- If there is no reference or product page and no help text associated with `name`, then `doc` searches the documentation for `name` and displays the search results in the Help browser.

When the browser is already open, the page displays in the most recently used tab.

Input Arguments

name - Name of function, class, block, product folder, or other functionality

string

Name of a function, class, block, product folder, or other functionality, specified as a string. Alternatively, an operator symbol.

Some classes require that you specify the package name. Events, properties, and some methods require that you specify the class name. Separate the components of the name with periods, using one of the following forms:

```
doc className.name
doc packageName.className
doc packageName.className.name
```

Methods for some classes are not accessible using the `doc` command; instead, use links on the class reference page.

If `name` is overloaded (that is, appears in multiple folders on the search path), then include a partial path to display the correct reference page, such as

```
doc folderName/name
```

Examples

Function Reference Pages

Display the reference page for the `abs` function.

```
doc abs
```

Several products include different versions of `abs`. If your Help preferences support displaying documentation for those products, then the Help browser displays the MATLAB `abs` reference page and a message with links to other versions of `abs`. This message appears at the top of the page.

Display the reference page for the `abs` function in the Fixed-Point Designer™ product by specifying the partial path.

```
doc fixedpoint/abs
```

Class and Method Reference Pages

Display the reference page for the `handle` class.

```
doc handle
```

Display the reference page for the `findobj` method in the `handle` class.

```
doc handle.findobj
```

Display the reference page for the `Map` class in the `containers` package.

```
doc containers.Map
```

Custom Class Pages

Display formatted help text for a custom class.

MATLAB includes a set of example files that show how to create a class, including a class file named `sads.m`. Add the example folder to the path, and request documentation for `sads`.

```
addpath(...
   fullfile(matlabroot,'help','techdoc','matlab_env','examples'))
doc sads
```

Display the help for the `steer` method of the `sads` class.

```
doc sads.steer
```

Because the help text follows MATLAB conventions, MATLAB formats the display in the Web browser.

Product Documentation

Display the main documentation page for a toolbox by specifying the toolbox folder name.

Identify the folder name for a product by calling `which`, using the name of a function unique to that product. For example, `ttest` is unique to the Statistics Toolbox product.

```
which ttest
```

```
matlabroot\toolbox\stats\stats\ttest.m
```

The value of `matlabroot` depends on your system. However, the product folder name always appears immediately after `toolbox` in the path.

Display the main page for the Statistics Toolbox.

```
doc stats
```

Tips

- To access third-party or custom documentation, open the Help browser and navigate to the documentation home page. Then, at the bottom of the page, click **Supplemental Software**.

See Also

`help` | `web`

Concepts

- “Ways to Get Function Help”
- “Add Help for Your Program”
- “Display Custom Documentation”

Purpose

Help browser search

Syntax

```
docsearch
docsearch expression
```

Description

`docsearch` opens the Help browser and displays the documentation home page. If the Help browser is already open, but not visible, then `docsearch` brings it to the foreground.

`docsearch expression` searches MathWorks documentation for pages with words that match the specified expression. The `docsearch` command does not search third-party or custom documentation.

Input Arguments

expression - Expression that defines search terms

string

Expression that defines search terms, specified as a string. Expressions can include:

- Quotation marks to specify exact phrases, such as `"plot tools"`.
- Boolean operator keywords in uppercase (listed here in order of precedence): `NOT`, `OR`, `AND`.
- Asterisk (`*`) wildcard characters, except at the beginning of a word or in an exact phrase. Searches require that at least two characters in the expression are *not* wildcard characters.

Examples

Single Words

Find all documentation pages that contain the word *plot*.

```
docsearch plot
```

Multiple Words

Find documentation pages with the words *plot* and *tools*.

```
docsearch plot tools
```

Expand the search to include variations of the word *plot*, such as *plotting* or *plots*, using a wildcard character.

```
docsearch plot* tools
```

Narrow the search to pages that include an exact phrase by enclosing the phrase in quotation marks.

```
docsearch "plot tools"
```

Find pages with either word, but not necessarily both words, using the `OR` operator.

```
docsearch plot OR tools
```

Tips

- To access third-party or custom documentation, open the Help browser and navigate to the documentation home page. Then, at the bottom of the page, click **Supplemental Software**.

See Also

`builddocsearchdb` | `doc`

Concepts

- "Search Syntax and Tips"

Purpose

Execute DOS command and return output

Syntax

```
status = dos(command)
[status,cmdout] = dos(command)
[status,cmdout] = dos(command,'-echo')
```

Description

`status = dos(command)` executes the specified DOS command for Windows platforms, and waits for the command to finish execution before returning the exit status to the `status` variable.

`[status,cmdout] = dos(command)` additionally returns the output of the DOS command to `cmdout`. This syntax is most useful for DOS console commands that do not require user input, such as `dir`.

`[status,cmdout] = dos(command,'-echo')` additionally displays (echoes) the command output in the MATLAB Command Window. This syntax is most useful for DOS console commands that require user input and that run correctly in the MATLAB Command Window, such as `comp`.

Input Arguments

command - MS-DOS® command

string

MS-DOS command, specified as a string. The command can be a Windows UI program that opens a graphical user interface, or a DOS console command that you typically run in a DOS command window. The command executes in a DOS shell, which might not be the shell from which you launched MATLAB.

Example: `'dir'`

Output Arguments

status - Command exit status

0 | nonzero integer

Command exit status, returned as either `0` or a nonzero integer. When the command is successful, `status` is `0`. Otherwise, `status` is a nonzero integer.

- If `command` includes the ampersand character (`&`), then `status` is the exit status upon command launch.
- If `command` does not include the ampersand character (`&`), then `status` is the exit status upon command completion.

cmdout - Output of operating system command

string

Output of the operating system command, returned as a string.

Limitations

- DOS does not support UNC path names. Therefore, if the current folder uses a UNC path name, then running `dos` with a DOS `command` that relies on the current folder fails. MATLAB returns this error:

  ```
  Error using dos
  DOS commands may not be executed when the current directory is a UNC pa
  ```

  To work around this limitation, change the folder to a mapped drive before running `dos` or a function that calls `dos`.

Examples

Save DOS Command Exit Status

Issue a DOS command to create a new folder named `mynew` and save the exit status to a variable.

```
command = 'mkdir mynew';
status = dos(command)

status =

     0
```

The `status` of zero indicates that the `mynew` folder was created successfully.

Open and Run a Windows UI Command

Open Notepad and immediately return the exit status to MATLAB by appending an ampersand (`&`) to the `notepad` command.

```
status = dos('notepad &')

status =

     0
```

The status of zero indicates that Notepad successfully started.

Save Successful DOS Command Status and Output

When you issue a valid DOS command, status indicates success and cmdout contains the command output.

```
[status,cmdout] = dos('dir');
status

status =

     0

cmdout

cmdout =

 Volume in drive C is OSDisk
 Volume Serial Number is XXX-XXXX

 Directory of C:\my_MATLAB_files

04/10/2012  12:08 PM <DIR>          .
04/10/2012  12:08 PM <DIR>          ..
04/21/2011  09:24 AM                171 base.mat
02/08/2010  05:14 PM                 73 baseball.dat
04/10/2012  12:08 PM                474 collatz.asv
04/10/2012  11:56 AM                480 collatz.m
.
.
.
```

Save Unsuccessful DOS Command Status and Output

When you issue an invalid DOS command, `status` indicates failure and `results` contains the DOS error message.

```
[status,results] = dos('foo');
status

status =

     1


results =

'foo' is not recognized as an internal or external command,
operable program or batch file.

results

results =

'foo' is not recognized as an internal or external command,
operable program or batch file.

>>
```

Display DOS Command Output in MATLAB Command Window

Display command output and prompts in the Command Window as the command executes, and also assign the command output to the `results` variable.

```
[status,results] = dos('comp', '-echo');

Name of first file to compare: collatz.m
collatz.m
```

```
Name of second file to compare: collatz.asv
collatz.asv
Option: /A
/A
Option:

Comparing collatz.m and collatz.asv...
Files compare OK

Compare more files (Y/N) ? N
N
>>
```

Tips

- To execute the operating system command in the background, include the trailing character, `&`, in the `command` argument (for example, `'notepad &'`). The exit status is immediately returned to the `status` variable. This syntax is useful for console programs that require interactive user command input while they run, and that do not run correctly in the MATLAB Command Window.

  Note If `command` includes the trailing `&` character, `cmdout` is empty.

See Also

`computer` | `perl` | `system` | `unix!` (exclamation point) |

Purpose

Vector dot product

Syntax

```
C = dot(A,B)
C = dot(A,B,dim)
```

Description

`C = dot(A,B)` returns the scalar product of the vectors `A` and `B`. `A` and `B` must be vectors of the same length. When `A` and `B` are both column vectors, `dot(A,B)` is the same as `A'*B`.

For multidimensional arrays `A` and `B`, `dot` returns the scalar product along the first non-singleton dimension of `A` and `B`. `A` and `B` must have the same size.

`C = dot(A,B,dim)` returns the scalar product of `A` and `B` in the dimension `dim`.

Examples

The dot product of two vectors is calculated as shown:

```
a = [1 2 3]; b = [4 5 6];
c = dot(a,b)

c =
      32
```

See Also

`cross`

Purpose

Convert to double precision

Syntax

```
double(x)
```

Description

`double(x)` returns the double-precision value for `X`. If `X` is already a double-precision array, `double` has no effect.

Tips

The `double` function should be overloaded for any object when it makes sense to convert it to a double-precision value.

dragrect

Purpose

Drag rectangles with mouse

Syntax

```
[finalrect] = dragrect(initialrect)
[finalrect] = dragrect(initialrect,stepsize)
```

Description

`[finalrect] = dragrect(initialrect)` tracks one or more rectangles anywhere on the screen. The n-by-4 matrix `initialrect` defines the rectangles. Each row of `initialrect` must contain the initial rectangle position as `[left bottom width height]` values. `dragrect` returns the final position of the rectangles in `finalrect`.

`[finalrect] = dragrect(initialrect,stepsize)` moves the rectangles in increments of `stepsize`. The lower left corner of the first rectangle is constrained to a grid of size equal to `stepsize` starting at the lower left corner of the figure, and all other rectangles maintain their original offset from the first rectangle.

`[finalrect] = dragrect(...)` returns the final positions of the rectangles when the mouse button is released. The default step size is `1`.

Tips

`dragrect` returns immediately if a mouse button is not currently pressed. Use `dragrect` in a `ButtonDownFcn`, or from the command line in conjunction with `waitforbuttonpress`, to ensure that the mouse button is down when `dragrect` is called. `dragrect` returns when you release the mouse button.

If the drag ends over a figure window, the positions of the rectangles are returned in that figure's coordinate system. If the drag ends over a part of the screen not contained within a figure window, the rectangles are returned in the coordinate system of the figure over which the drag began.

Note You cannot use normalized figure units with `dragrect`.

Examples

Drag a rectangle that is 50 pixels wide and 100 pixels in height.

```
waitforbuttonpress
point1 = get(gcf,'CurrentPoint') % button down detected
rect = [point1(1,1) point1(1,2) 50 100]
[r2] = dragrect(rect)
```

See Also

rbbox | waitforbuttonpress

drawnow

Purpose

Flush event queue and update figure window

Syntax

```
drawnow
drawnow expose
drawnow update
```

Description

`drawnow` causes figure windows and their children to update, and flushes the system event queue. Any callbacks generated by incoming events (e.g., mouse or key events) are dispatched before `drawnow` returns.

`drawnow expose` causes only graphics objects to refresh, if needed. It does not allow callbacks to execute and does not process other events in the queue.

`drawnow update` causes only non graphics objects to refresh, if needed. It does not allow callbacks to execute and does not process other events in the queue.

You can combine the `expose` and `update` options to obtain both effects:

```
drawnow expose update
```

Other Events That Cause Event Queue Processing

Other events that cause the MATLAB software to flush the event queue and draw the figure include:

- Returning to the MATLAB prompt
- Executing the following functions:
  - `figure`
  - `getframe`
  - `input`
  - `keyboard`
- Functions that wait for user input (i.e., `waitforbuttonpress`, `waitfor`, `ginput`)

- Any code that causes one of the above functions to execute. For example, suppose `h` is the handle of an axes. Calling `axes(h)` causes its parent figure to be made the current figure and brought to the front of all displayed figures, which results in the event queue being flushed.

Examples

Using `drawnow` in a loop causes the display to update while the loop executes:

```
t = 0:pi/20:2*pi;
y = exp(sin(t));
h = plot(t,y,'YDataSource','y');
for k = 1:.1:10
 y = exp(sin(t.*k));
 refreshdata(h,'caller') % Evaluate y in the function workspace
 drawnow; pause(.1)
end
```

See Also

`snapnow` | `waitfor` | `waitforbuttonpress`

dsearchn

Purpose

N-D nearest point search

Syntax

```
k = dsearchn(X,T,XI)
k = dsearchn(X,T,XI,outval)
k = dsearchn(X,XI)
[k,d] = dsearchn(X,...)
```

Description

`k = dsearchn(X,T,XI)` returns the indices `k` of the closest points in `X` for each point in `XI`. `X` is an `m`-by-`n` matrix representing `m` points in n-dimensional space. `XI` is a `p`-by-`n` matrix, representing `p` points in n-dimensional space. `T` is a `numt`-by-`n+1` matrix, a triangulation of the data `X` generated by `delaunayn`. The output `k` is a column vector of length `p`.

`k = dsearchn(X,T,XI,outval)` returns the indices `k` of the closest points in `X` for each point in `XI`, unless a point is outside the convex hull. If `XI(J,:)` is outside the convex hull, then `K(J)` is assigned `outval`, a scalar double. `Inf` is often used for `outval`. If `outval` is `[]`, then `k` is the same as in the case `k = dsearchn(X,T,XI)`.

`k = dsearchn(X,XI)` performs the search without using a triangulation. With large `X` and small `XI`, this approach is faster and uses much less memory.

`[k,d] = dsearchn(X,...)` also returns the distances `d` to the closest points. `d` is a column vector of length `p`.

Algorithms

`dsearchn` is based on Qhull [1]. For information about Qhull, see `http://www.qhull.org/`. For copyright information, see `http://www.qhull.org/COPYING.txt`.

References

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," ACM Transactions on Mathematical Software, Vol. 22, No. 4, Dec. 1996, p. 469–483.

See Also

`delaunayTriangulation`

Purpose

Abstract class used to derive handle class with dynamic properties

Syntax

`classdef myclass < dynamicprops`

Description

`classdef myclass < dynamicprops` makes *`myclass`* a subclass of the `dynamicprops` class, which is a subclass of the `handle` class.

Use the `dynamicprops` class to derive classes that can define dynamic properties (instance properties), which are associated with a specific objects, but have no effect on the objects class definition. Dynamic properties are useful for attaching temporary data to one or more objects.

dynamicprops Methods

This class defines one method `addprop` and, as a subclass of the `handle` class, inherits all the `handle` class methods.

- `addprop` — adds the named property to the specified handle objects. See “Dynamic Properties — Adding Properties to an Instance” for more information.

See Also

`handle`

echo

Purpose

Display statements during function execution

Syntax

```
echo on
echo off
echo
echo fcnname on
echo fcnname off
echo fcnname
echo on all
echo off all
```

Description

The `echo` command controls the display (or *echoing*) of statements in a function during their execution. Normally, statements in a function file are not displayed on the screen during execution. Command echoing is useful for debugging or for demonstrations, allowing the commands to be viewed as they execute.

The `echo` command behaves in a slightly different manner for script files and function files. For script files, the use of `echo` is simple; echoing can be either `on` or `off`, in which case any script used is affected.

`echo on`	Turns on the echoing of commands in all script files
`echo off`	Turns off the echoing of commands in all script files
`echo`	Toggles the echo state

With function files, the use of `echo` is more complicated. If `echo` is enabled on a function file, the file is interpreted, rather than compiled. Each input line is then displayed as it is executed. Since this results in inefficient execution, use `echo` only for debugging.

`echo` *`fcnname`* `on`	Turns on echoing of the named function file
`echo` *`fcnname`* `off`	Turns off echoing of the named function file

`echo` *`fcnname`*	Toggles the echo state of the named function file
`echo on all`	Sets echoing on for all function files
`echo off all`	Sets echoing off for all function files

Tips

- To avoid confusing syntax, do not use `on` or `off` as a function name.

See Also

`function`

Purpose

Run example script step-by-step in Command Window

Syntax

```
echodemo filename
echodemo(filename,index)
```

Description

`echodemo filename` runs the script specified by `filename` step-by-step in the Command Window. The file must contain sections defined with two percent signs (`%%`) to enable pausing after each step. At each step, you can click links in the Command Window to proceed or stop. If the Command Window is not large enough to show the links, scroll up to see them.

`echodemo(filename,index)` starts with the section number specified by `index`. If the example relies on results of previous steps, using this syntax can produce errors or unexpected results.

Caution

If variables in your base workspace have the same name as variables that the example file creates, the example could overwrite your data. Preserve your data by saving it to a MAT-file before running the example.

Input Arguments

filename - Script file name
string

Script file name, specified as a string.

When you use the function syntax for `echodemo` and specify its inputs within parentheses, enclose the `filename` input in single quotes.

index - Section index
scalar integer

Section index, specified as a scalar integer.

The link text in the Command Window shows the current section number, `n`, and the total number of sections, `m`, as `n/m`.

Examples

Run Example Script in Command Window

Run the Loma Prieta Earthquake example.

```
echodemo quake
```

Start Script from Specified Section

Start the Loma Prieta Earthquake example from the third section.

```
filename = 'quake';
index = 3;
echodemo(filename,index)
```

This code errors because the example requires variables created in earlier sections.

Tips

- Only use `echodemo` to display scripts, not functions. `echodemo` can run any script that you can execute, but only scripts with sections pause between steps.

See Also

`demo` | `doc` | `publish`

TriRep.edgeAttachments

Purpose

(Will be removed) Simplices attached to specified edges

Note `edgeAttachments(TriRep)` will be removed in a future release. Use `edgeAttachments(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
SI = edgeAttachments(TR, V1, V2)
SI = edgeAttachments(TR, EDGE)
```

Description

`SI = edgeAttachments(TR, V1, V2)` returns the simplices `SI` attached to the edges specified by `(V1, V2)`. `(V1, V2)` represents the start and end vertices of the edges to be queried.

`SI = edgeAttachments(TR, EDGE)` specifies edges in matrix format.

Input Arguments

`TR`	Triangulation representation.
`V1,V2`	Column vectors of vertex indices into the array of points representing the vertex coordinates.
`EDGE`	Matrix specifying edge start and end points. `EDGE` is of size `m`-by-2, `m` being the number of edges to query.

Output Arguments

`SI`	Vector cell array of indices into the triangulation matrix. `SI` is a cell array because the number of simplices associated with each edge can vary.

Definitions

A simplex is a triangle/tetrahedron or higher dimensional equivalent.

Examples

Example 1

Load a 3-D triangulation to compute the tetrahedra attached to an edge.

```
load tetmesh
trep = TriRep(tet, X);
v1 = [15 21]';
v2 = [936 716]';
t1 = edgeAttachments(trep, v1, v2);
```

You can also specify the input as edges.

```
e = [v1 v2];
t2 = edgeAttachments(trep, e);
isequal(t1,t2);
```

Example 2

Create a triangulation with `DelaunayTri`.

```
x = [0 1 1 0 0.5]';
y = [0 0 1 1 0.5]';
dt = DelaunayTri(x,y);
```

Query the triangles attached to edge (1,5).

```
t = edgeAttachments(dt, 1,5);
t{:};
```

See Also

`edges` | `triangulation` | `delaunayTriangulation`

TriRep.edges

Purpose

(Will be removed) Triangulation edges

Note `edges(TriRep)` will be removed in a future release. Use `edges(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
E = edges(TR)
```

Description

`E = edges(TR)` returns the edges in the triangulation in an `n`-by-2 matrix. `n` is the number of edges. The vertices of the edges index into `TR.X`, the array of points representing the vertex coordinates.

Input Arguments

`TR`	Triangulation representation.

Output Arguments

`E`	Edge matrix.

Examples

Example 1

Load a 2-D triangulation.

```
load trimesh2d
trep = TriRep(tri, x,y);
```

Return all edges.

```
e = edges(trep);
```

Example 2

Query a 2-D `DelaunayTri`-generated triangulation.

```
X = rand(10,2);
```

```
dt = DelaunayTri(X);
e = edges(dt);
```

See Also

edgeAttachments | triangulation | delaunayTriangulation

Purpose

Edit or create file

Syntax

```
edit
edit file
edit file1 ... fileN
```

Description

`edit` opens a new file called `Untitled` in the Editor. MATLAB does not automatically save `Untitled`.

`edit file` opens the specified file in the Editor. If `file` does not already exist, MATLAB asks if you want to create it. `file` can include a partial path, complete path, relative path, or no path. You must have write permission to the path to create `file`, otherwise, MATLAB ignores the argument.

You must specify the extension to open `.mat` and `.mdl` files. MATLAB cannot directly edit binary files, such as `.p` and `.mex` files.

`edit file1 ... fileN` opens each file, `file1 ... fileN`, in the Editor.

Examples

Open a New File

```
edit
```

A new file titled `Untitled` opens in the MATLAB Editor (or default editor). `Untitled` does not appear in your Current Folder.

Create a New File

```
mkdir tests
edit tests/new_script.m
```

A dialog box appears, asking if you want to create `new_script.m`. If you select **Yes**, MATLAB creates and opens `tests/new_script.m`.

Open Files

```
edit file1 file2 file3 file4
```

MATLAB sequentially creates and opens the files: `file1`, `file2`, `file3`, and `file4` in sequence.

Input Arguments

file - Name of file

string

Name of file, specified as a string. If `file` specifies a path that contains a nonexistent folder, MATLAB throws an error.

Data Types

`char`

file1 ... fileN - Names of files

strings

Names of files, specified as strings. You can specify `file1 ... fileN` as a mixture of partial paths, complete paths, relative paths, or no paths.

Data Types

`char`

See Also

`open | type`

Concepts

- "Change Default Editor"

Purpose

Eigenvalues and eigenvectors

Syntax

```
d = eig(A)
d = eig(A,B)
[V,D] = eig(A)
[V,D] = eig(A,'nobalance')
[V,D] = eig(A,B)
[V,D] = eig(A,B,flag)
```

Description

`d = eig(A)` returns a vector of the eigenvalues of matrix `A`.

`d = eig(A,B)` returns a vector containing the generalized eigenvalues, if `A` and `B` are square matrices.

Note If `S` is sparse and symmetric, you can use `d = eig(S)` to return the eigenvalues of `S`. If `S` is sparse but not symmetric, or if you want to return the eigenvectors of `S`, use the function `eigs` instead of `eig`.

`[V,D] = eig(A)` produces matrices of eigenvalues (`D`) and eigenvectors (`V`) of matrix `A`, so that `A*V = V*D`. Matrix `D` is the *canonical form* of `A` — a diagonal matrix with `A`'s eigenvalues on the main diagonal. Matrix `V` is the *modal matrix* — its columns are the eigenvectors of `A`.

`[V,D] = eig(A,'nobalance')` finds eigenvalues and eigenvectors without a preliminary balancing step. This may give more accurate results for certain problems with unusual scaling. Ordinarily, balancing improves the conditioning of the input matrix, enabling more accurate computation of the eigenvectors and eigenvalues. However, if a matrix contains small elements that are really due to roundoff error, balancing may scale them up to make them as significant as the other elements of the original matrix, leading to incorrect eigenvectors. Use the `nobalance` option in this event. See the `balance` function for more details.

`[V,D] = eig(A,B)` produces a diagonal matrix `D` of generalized eigenvalues and a full matrix `V` whose columns are the corresponding eigenvectors so that `A*V = B*V*D`.

`[V,D] = eig(A,B,flag)` specifies the algorithm used to compute eigenvalues and eigenvectors. *`flag`* can be:

`'chol'`	Computes the generalized eigenvalues of `A` and `B` using the Cholesky factorization of `B`. This is the default for symmetric (Hermitian) `A` and symmetric (Hermitian) positive definite `B`.
`'qz'`	Ignores the symmetry, if any, and uses the QZ algorithm as it would for nonsymmetric (non-Hermitian) `A` and `B`.

Note For `eig(A)`, the eigenvectors are scaled so that the norm of each is 1.0. For `eig(A,B)`, `eig(A,'nobalance')`, and `eig(A,B,flag)`, the eigenvectors are not normalized.

Also note that if `A` is symmetric, `eig(A,'nobalance')` ignores the `nobalance` option since `A` is already balanced.

Tips

The eigenvalue problem is to determine the nontrivial solutions of the equation $Ax = \lambda x$

where A is an n-by-n matrix, x is a length n column vector, and λ is a scalar. The n values of λ that satisfy the equation are the *eigenvalues*, and the corresponding values of x are the *right eigenvectors*. The MATLAB function `eig` solves for the eigenvalues λ, and optionally the eigenvectors x.

The *generalized* eigenvalue problem is to determine the nontrivial solutions of the equation $Ax = \lambda Bx$

where both A and B are n-by-n matrices and λ is a scalar. The values of λ that satisfy the equation are the *generalized eigenvalues* and the corresponding values of x are the *generalized right eigenvectors*.

If B is nonsingular, the problem could be solved by reducing it to a standard eigenvalue problem $B^{-1}Ax = \lambda x$

Because B can be singular, an alternative algorithm, called the QZ method, is necessary.

When a matrix has no repeated eigenvalues, the eigenvectors are always independent and the eigenvector matrix V *diagonalizes* the original matrix A if applied as a similarity transformation. However, if a matrix has repeated eigenvalues, it is not similar to a diagonal matrix unless it has a full (independent) set of eigenvectors. If the eigenvectors are not independent then the original matrix is said to be *defective*. Even if a matrix is defective, the solution from eig satisfies A*X = X*D.

Examples

The matrix

```
B = [ 3     -2      -.9    2*eps
     -2      4       1    -eps
     -eps/4  eps/2  -1     0
     -.5    -.5      .1    1   ];
```

has elements on the order of roundoff error. It is an example for which the nobalance option is necessary to compute the eigenvectors correctly. Try the statements

```
[VB,DB] = eig(B)
B*VB - VB*DB
[VN,DN] = eig(B,'nobalance')
B*VN - VN*DN
```

See Also

balance | condeig | eigs | hess | qz | schur

Purpose

Largest eigenvalues and eigenvectors of matrix

Syntax

```
d = eigs(A)
[V,D] = eigs(A)
[V,D,flag] = eigs(A)
eigs(A,B)
eigs(A,k)
eigs(A,B,k)
eigs(A,k,sigma)
eigs(A,B,k,sigma)
eigs(A,K,sigma,opts)
eigs(A,B,k,sigma,opts)
eigs(Afun,n,...)
```

Description

`d = eigs(A)` returns a vector of `A`'s six largest magnitude eigenvalues. `A` must be a square matrix. `A` should be large and sparse, though `eigs` will work on full matrices as well. See "Tips" below.

`[V,D] = eigs(A)` returns a diagonal matrix `D` of `A`'s six largest magnitude eigenvalues and a matrix `V` whose columns are the corresponding eigenvectors.

`[V,D,flag] = eigs(A)` also returns a convergence flag. If `flag` is `0` then all the eigenvalues converged; otherwise not all converged.

`eigs(A,B)` solves the generalized eigenvalue problem `A*V == B*V*D`. `B` must be the same size as `A`. `eigs(A,[],...)` indicates the standard eigenvalue problem `A*V == V*D`.

`eigs(A,k)` and `eigs(A,B,k)` return the `k` largest magnitude eigenvalues.

`eigs(A,k,`*`sigma`*`)` and `eigs(A,B,k,`*`sigma`*`)` return `k` eigenvalues based on *`sigma`*, which can take any of the following values:

scalar (real or complex, including 0)	The eigenvalues closest to `sigma`. If `A` is a function, `Afun` must return `Y = (A-sigma*B)\x` (i.e., `Y = A\x` when `sigma = 0`).
'lm'	Largest magnitude (default).
'sm'	Smallest magnitude. Same as `sigma = 0`. If `A` is a function, `Afun` must return `Y = A\x`.

For real symmetric problems, the following are also options:

'la'	Largest algebraic ('lr')
'sa'	Smallest algebraic ('sr')
'be'	Both ends (one more from high end if `k` is odd)

For nonsymmetric and complex problems, the following are also options:

'lr'	Largest real part
'sr'	Smallest real part
'li'	Largest imaginary part
'si'	Smallest imaginary part

Note The syntax `eigs(A,k,...)` is not valid when `A` is scalar. To pass a value for `k`, you must specify `B` as the second argument and `k` as the third (`eigs(A,B,k,...)`). If necessary, you can set `B` equal to `[]`, the default.

`eigs(A,K,sigma,opts)` and `eigs(A,B,k,sigma,opts)` specify an options structure. Default values are shown in brackets (`{}`).

Parameter	Description	Values
opts.issym	1 if A or A-*sigma**B represented by Afun is symmetric, 0 otherwise.	[{0}	1]
opts.isreal	1 if A or A-*sigma**B represented by Afun is real, 0 otherwise.	[0	{1}]
opts.tol	Convergence: Ritz estimate residual <= tol*norm(A).	[scalar	{eps}]
opts.maxit	Maximum number of iterations.	[integer	{300}]
opts.p	Number of Lanczos basis vectors. p >= 2k (p >= 2k+1 real nonsymmetric) advised. p must satisfy k < p <= n for real symmetric, k+1 < p <= n otherwise. Note: If you do not specify a p value, the default algorithm uses at least 20 Lanczos vectors.	[integer	{2*k}]
opts.v0	Starting vector.	[n-by-1 vector	{randomly generated by rand}]
opts.disp	Diagnostic information display level.	[{0}	1	2]
opts.cholB	1 if B is really its Cholesky factor chol(B), 0 otherwise.	[{0}	1]
opts.permB	Permutation vector permB if sparse B is really chol(B(permB,permB)).	[permB	{1:n}]

`eigs(Afun,n,...)` accepts a function handle, `Afun`, instead of the matrix `A`.

`y = Afun(x)` should return:

`A*x`	if *sigma* is not specified, or is a string other than `'sm'`
`A\x`	if *sigma* is `0` or `'sm'`
`(A-sigma*I)\x`	if *sigma* is a nonzero scalar (standard eigenvalue problem). `I` is an identity matrix of the same size as `A`.
`(A-sigma*B)\x`	if *sigma* is a nonzero scalar (generalized eigenvalue problem)

"Parameterizing Functions" explains how to provide additional parameters to the function `Afun`, if necessary.

The matrix `A`, `A-sigma*I` or `A-sigma*B` represented by `Afun` is assumed to be real and nonsymmetric unless specified otherwise by `opts.isreal` and `opts.issym`. In all the `eigs` syntaxes, `eigs(A,...)` can be replaced by `eigs(Afun,n,...)`.

Tips

`d = eigs(A,k)` is not a substitute for

```
d = eig(full(A))
d = sort(d)
d = d(end-k+1:end)
```

but is most appropriate for large sparse matrices. If the problem fits into memory, it may be quicker to use `eig(full(A))`.

Unless you provide a start vector with `opts.v0`, the default start vector is generated by `rand`, possibly leading to different iterations each run, and perhaps even different convergence behavior. In order to control this, specify your start vector via `opts.v0`.

Examples

Example 1

```
A = delsq(numgrid('C',15));
d1 = eigs(A,5,'sm')
```

returns

```
d1 =
    0.5520
    0.4787
    0.3469
    0.2676
    0.1334
```

Example 2

This example replaces the matrix `A` in example 1 with a handle to a function `dnRk`. The example is contained in file `run_eigs` that

- Calls `eigs` with the function handle `@dnRk` as its first argument.
- Contains `dnRk` as a nested function, so that all variables in `run_eigs` are available to `dnRk`.

The following shows the code for `run_eigs`:

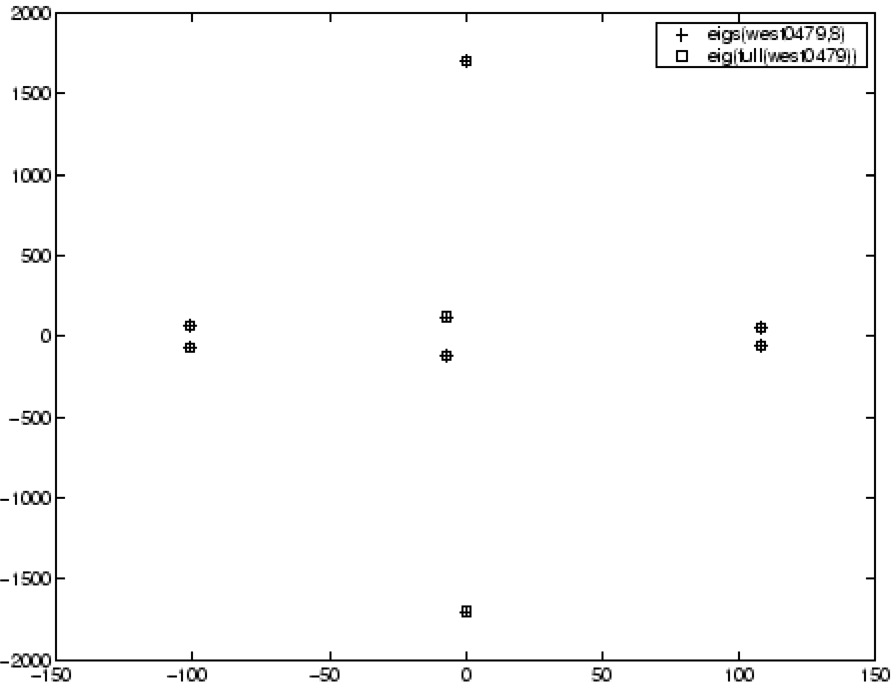
```
function d2 = run_eigs
n = 139;
opts.issym = 1;
R = 'C';
k = 15;
d2 = eigs(@dnRk,n,5,'sm',opts);

    function y = dnRk(x)
        y = (delsq(numgrid(R,k))) \ x;
    end
end
```

Example 3

west0479 is a real 479-by-479 sparse matrix with both real and pairs of complex conjugate eigenvalues. eig computes all 479 eigenvalues. eigs easily picks out the largest magnitude eigenvalues.

This plot shows the 8 largest magnitude eigenvalues of west0479 as computed by eig and eigs.

```
load west0479
d = eig(full(west0479))
dlm = eigs(west0479,8)
[dum,ind] = sort(abs(d));
plot(dlm,'k+')
hold on
plot(d(ind(end-7:end)),'ks')
hold off
legend('eigs(west0479,8)','eig(full(west0479))')
```



Example 4

A = delsq(numgrid('C',30)) is a symmetric positive definite matrix of size 632 with eigenvalues reasonably well-distributed in the interval (0 8), but with 18 eigenvalues repeated at 4. The eig function computes all 632 eigenvalues. It computes and plots the six largest and smallest magnitude eigenvalues of A successfully with:

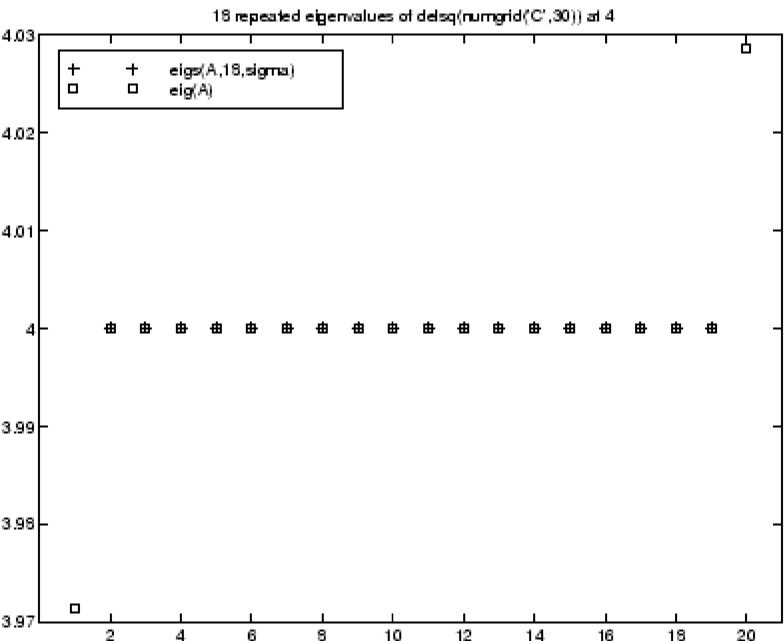
```
A = delsq(numgrid('C',30));
d = eig(full(A));
[dum,ind] = sort(abs(d));
dlm = eigs(A);
dsm = eigs(A,6,'sm');

subplot(2,1,1)
plot(dlm,'k+')
```

```
hold on
plot(d(ind(end:-1:end-5)),'ks')
hold off
legend('eigs(A)','eig(full(A))',3)
set(gca,'XLim',[0.5 6.5])

subplot(2,1,2)
plot(dsm,'k+')
hold on
plot(d(ind(1:6)),'ks')
hold off
legend('eigs(A,6,''sm'')','eig(full(A))',2)
set(gca,'XLim',[0.5 6.5])
```

However, the repeated eigenvalue at 4 must be handled more carefully. The call `eigs(A,18,4.0)` to compute 18 eigenvalues near 4.0 tries to find eigenvalues of `A - 4.0*I`. This involves divisions of the form `1/(lambda - 4.0)`, where `lambda` is an estimate of an eigenvalue of `A`. As `lambda` gets closer to 4.0, `eigs` fails. We must use `sigma` near but not equal to 4 to find those 18 eigenvalues.

```
sigma = 4 - 1e-6
[V,D] = eigs(A,18,sigma)
```

The plot shows the 20 eigenvalues closest to 4 that were computed by `eig`, along with the 18 eigenvalues closest to `4 - 1e-6` that were computed by `eigs`.



References

[1] Lehoucq, R.B. and D.C. Sorensen, "Deflation Techniques for an Implicitly Re-Started Arnoldi Iteration," *SIAM J. Matrix Analysis and Applications*, Vol. 17, 1996, pp. 789–821.

[2] Sorensen, D.C., "Implicit Application of Polynomial Filters in a k-Step Arnoldi Method," *SIAM J. Matrix Analysis and Applications*, Vol. 13, 1992, pp. 357–385.

See Also

`eig` | `svds` | `function_handle`

Purpose

Jacobi elliptic functions

Syntax

```
[SN,CN,DN] = ellipj(U,M)
[SN,CN,DN] = ellipj(U,M,tol)
```

Definitions

The Jacobi elliptic functions are defined in terms of the integral:

$$u = \int_0^{\phi} \frac{d\theta}{\sqrt{1 - m\sin^2\theta}}.$$

Then

$$sn(u) = \sin\phi,\ cn(u) = \cos\phi,\ dn(u) = \sqrt{1 - m\sin^2\phi}.$$

Some definitions of the elliptic functions use the modulus k instead of the parameter m. They are related by

$$k^2 = m = \sin^2 a,$$

where a is the modular angle.

The Jacobi elliptic functions obey many mathematical identities; for a good sample, see [1].

Description

`[SN,CN,DN] = ellipj(U,M)` returns the Jacobi elliptic functions `SN`, `CN`, and `DN`, evaluated for corresponding elements of argument `U` and parameter `M`. Inputs `U` and `M` must be the same size (or either can be scalar).

`[SN,CN,DN] = ellipj(U,M,tol)` computes the Jacobi elliptic functions to accuracy `tol`. The default is `eps`; increase this for a less accurate but more quickly computed answer.

Algorithms

`ellipj` computes the Jacobi elliptic functions using the method of the arithmetic-geometric mean [1]. It starts with the triplet of numbers:

$$a_0 = 1,\ b_0 = \sqrt{1-m},\ c_0 = \sqrt{m}$$

`ellipj` computes successive iterates with

$$a_i = \frac{1}{2}(a_{i-1} + b_{i-1})$$
$$b_i = (a_{i-1}b_{i-1})^{\frac{1}{2}}$$
$$c_i = \frac{1}{2}(a_{i-1} - b_{i-1})$$

Next, it calculates the amplitudes in radians using:

$$\sin(2\phi_{n-1} - \phi_n) = \frac{c_n}{a_n}\sin(\phi_n)$$

being careful to unwrap the phases correctly. The Jacobian elliptic functions are then simply:

$$sn(u) = \sin\phi_0$$
$$cn(u) = \cos\phi_0$$
$$dn(u) = \sqrt{1 - m \cdot sn(u)^2}$$

Limitations

The `ellipj` function is limited to the input domain $0 \le m \le 1$. Map other values of `M` into this range using the transformations described in [1], equations 16.10 and 16.11. `U` is limited to real values.

References

[1] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, 17.6.

See Also

`ellipke`

Purpose

Complete elliptic integrals of first and second kind

Syntax

```
K = ellipke(M)
[K,E] = ellipke(M)
[K,E] = ellipke(M,tol)
```

Description

`K = ellipke(M)` returns the complete elliptic integral of the first kind for the each element in `M`.

`[K,E] = ellipke(M)` returns the complete elliptic integral of the first and second kinds.

`[K,E] = ellipke(M,tol)` computes the complete elliptic integral to accuracy `tol`. The default is `eps(class(M))`; increase the tolerance for a less accurate but more quickly computed answer.

Limitations

`ellipke` is limited to the input domain $0 \le m \le 1$.

Definitions

The *complete* elliptic integral of the first kind is

$$[K(m)] = \int_0^1 [(1-t^2)(1-mt^2)]^{-\frac{1}{2}} dt.$$

where m is the first argument of `ellipke`.

The complete elliptic integral of the second kind is

$$E(m) = \int_0^1 (1-t^2)^{-\frac{1}{2}} (1-mt^2)^{\frac{1}{2}} dt.$$

Some definitions of `K` and `E` use the elliptical modulus k instead of the parameter m. They are related as

$$k^2 = m = \sin^2 \alpha.$$

References

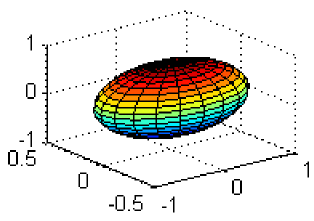
[1] Abramowitz, M., and I.A. Stegun. *Handbook of Mathematical Functions*. Dover Publications, 1965.

See Also

`ellipj`

Purpose

Generate ellipsoid



Syntax

```
[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr,n)
[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr)
ellipsoid(axes_handle,...)
ellipsoid(...)
```

Description

`[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr,n)` generates a surface mesh described by three `n+1`-by-`n+1` matrices, enabling `surf(x,y,z)` to plot an ellipsoid with center `(xc,yc,zc)` and semi-axis lengths `(xr,yr,zr)`.

`[x,y,z] = ellipsoid(xc,yc,zc,xr,yr,zr)` uses `n = 20`.

`ellipsoid(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`ellipsoid(...)` with no output arguments plots the ellipsoid as a surface.

Algorithms

`ellipsoid` generates the data using the following equation:

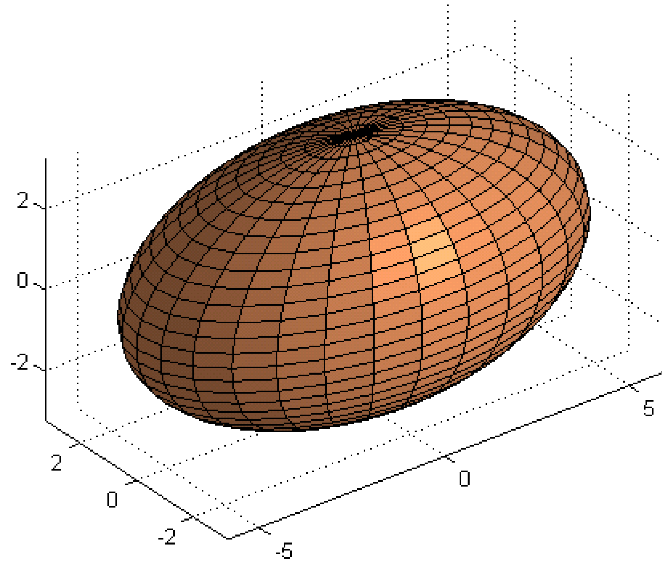
$$\frac{(x-xc)^2}{xr^2}+\frac{(y-yc)^2}{yr^2}+\frac{(z-zc)^2}{zr^2}$$

Note that `ellipsoid(0,0,0, .5,.5,.5)` is equivalent to a unit sphere.

Examples

Generate ellipsoid with size and proportions of a standard U.S. football:

```
[x, y, z] = ellipsoid(0,0,0,5.9,3.25,3.25,30);
surfl(x, y, z)
colormap copper
axis equal
```



See Also

cylinder | sphere | surf

Purpose

Optional keyword within an if statement

Syntax

```
if expression
   statements
elseif expression
   statements
else
   statements
end
```

Description

An `if` statement evaluates an expression, and executes a group of statements when the expression is true.

`elseif` and `else` are optional, and execute only when previous expressions in the `if` statement are false.

For more information about valid expressions and when they are true, see the `if` reference page.

Examples

Find the indices of values in a vector that are greater than a specified limit:

```
A = rand(1,10);
limit = .75;

B = (A > limit);   % B is a vector of logical values
if any(B)
  fprintf('Indices of values > %4.2f: \n', limit);
  disp(find(B))
else
  disp('All values are below the limit.')
end
```

See Also

`if` | `end` | `for` | `while` | `switch`

elseif

Purpose

Optional keyword within an if statement

Syntax

```
if expression
   statements
elseif expression
   statements
else
   statements
end
```

Description

An `if` statement evaluates an expression, and executes a group of statements when the expression is true.

`elseif` and `else` are optional, and execute only when previous expressions in the `if` statement are false.

For more information about valid expressions and when they are true, see the `if` reference page.

Tips

- Avoid adding a space within the `elseif` keyword (`else if`). The space creates a nested `if` statement that requires its own `end` keyword.

Examples

Assign to a matrix values that depend on their indices:

```
% Preallocate a matrix
nrows = 10;
ncols = 10;
myData = ones(nrows, ncols);

% Loop through the matrix
for r = 1:nrows
   for c = 1:ncols

      if r == c
         myData(r,c) = 2;
      elseif abs(r - c) == 1
```

```
            myData(r,c) = -1;
         else
            myData(r,c) = 0;
         end

      end
   end
```

See Also

if | end | for | while | switch

empty

Purpose

Create empty array

Syntax

```
A = ClassName.empty
A = ClassName.empty(n,m,p,...)
A = ClassName.empty([n,m,p,...])
```

Description

Use `empty` to create empty arrays of the specified class, *`ClassName`*. Specify at lease one dimension of the array as 0. MATLAB treats negative values as 0.

`A = ClassName.empty` returns an empty 0-by-0 array of the class of *`ClassName`*.

`A = ClassName.empty(n,m,p,...)` returns an empty rectangular array with the specified dimensions. At least one of the dimensions must be 0.

`A = ClassName.empty([n,m,p,...])` returns an empty rectangular array with the specified dimensions. At least one of the dimensions must be 0. This syntax is useful when using the values returned by the `size` function to define an empty array that is the same size as an existing empty array:

```
A = ClassName.empty(size(otherEmptyArray));
```

Input Arguments

n,m,p,...

Dimensions of the empty array. At least one of the specified dimensions must be 0.

Output Arguments

A

An empty array of the specified dimensions and of the class used in the method invocation.

Attributes

`empty` is a hidden, public, static method of all nonabstract MATLAB classes.

Access	Public
Hidden	true
Static	true

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Use `empty` to create a rectangular empty array of class `int16`:

```
A = int16.empty(5,0);
whos
  Name      Size            Bytes  Class    Attributes

  A         5x0                 0  int16
```

Using the `empty` method of the `int16` class to produce an empty array in which some dimensions are not zero is simpler than using conversion and reshape operations:

```
A = int16([]);
A = reshape(A,5,0);
whos
  Name      Size            Bytes  Class    Attributes

  A         5x0                 0  int16
```

Given the following definition for a class,

```
classdef ExEmpty
   properties
      Color = [1,0,0];
   end
   methods
      function obj = ExEmpty(c)
         if nargin > 0
            obj.Color = c;
         end
```

```
      end
   end
end
```

Create an empty array of class `ExEmpty`:

```
A = ExEmpty.empty;
 whos
  Name          Size              Bytes  Class          Attributes

  A             0x0               104  ExEmpty
```

One dimension of an empty array must be zero:

```
A5 = ExEmpty.empty(0,5);
 whos
  Name          Size              Bytes  Class          Attributes

  A5             0x5               104  ExEmpty
```

Empty object arrays follow array concatenation behavior:

```
B = [A,A5]
B =

  0x5 ExEmpty array with properties:

    Color
```

You cannot index into an empty array:

```
A5(1)
Index exceeds matrix dimensions.
```

You can use the `isempty`, `size`, and `length` functions to identify empty object arrays:

```
isempty(A5)
```

```
ans =

     1
size(A5)

ans =

     0     5
length(A5)

ans =

     0
```

Class of Empty Object Array

The `empty` method enables you to initialize arrays of a specific class:

```
C = char.empty(0,7)

C =

   Empty string: 0-by-7

class(C)
ans =

char
```

Initializing an array with empty brackets (`[]`):

```
a = [];
```

produces an array of class `double`:

```
class(a)
ans =
```

`double`

See Also

`| isempty | size | length`

Tutorials

- "Creating Empty Arrays"
- "Empty Matrices, Scalars, and Vectors"

Purpose

Enable access to .NET commands from network drive

Syntax

```
enableNETfromNetworkDrive
```

Description

`enableNETfromNetworkDrive` adds an entry for the MATLAB interface to .NET module to the security policy on your machine. You must have administrative privileges to make changes to your configuration.

Compatibility

Use `enableNETfromNetworkDrive` for MATLAB releases R2012b or earlier, which support installed versions 2.0, 3.0, and 3.5 of the Microsoft .NET Framework.

How To

- "Troubleshooting Security Policy Settings From Network Drives"

enableservice

Purpose

Enable, disable, or report status of MATLAB Automation server

Syntax

```
state = enableservice('AutomationServer',enable)
state = enableservice('AutomationServer')
```

Description

`state = enableservice('AutomationServer',enable)` enables or disables the MATLAB Automation server. If `enable` is `true` (logical `1`), `enableservice` converts an existing MATLAB session into an Automation server. If `enable` is `false` (logical `0`), `enableservice` disables the MATLAB Automation server. `state` indicates the previous state of the Automation server. If `state` = `1`, MATLAB was an Automation server. If `state` = `0`, MATLAB was not an Automation server.

`state = enableservice('AutomationServer')` returns the current state of the Automation server. If `state` is logical `1` (`true`), MATLAB is an Automation server.

COM functions are available on Microsoft Windows systems only.

Examples

Enable the Automation server in the current MATLAB session:

```
state = enableservice('AutomationServer',true);
```

Show the current state of the MATLAB session. MATLAB displays `true`:

```
state = enableservice('AutomationServer')
```

Enable the Automation server and show the previous state. MATLAB displays `true`. The previous state can be the same as the current state:

```
state = enableservice('AutomationServer',true)
```

See Also

`actxserver`

How To

- “MATLAB COM Automation Server Interface”

Purpose

Terminate block of code, or indicate last array index

Syntax

```
end
```

Description

`end` terminates `for`, `while`, `switch`, `try`, `if`, and `parfor` statements. Without an `end` statement, `for`, `while`, `switch`, `try`, `if`, and `parfor` wait for further input. Each `end` is paired with the closest previous unpaired `for`, `while`, `switch`, `try`, `if`, or `parfor` and serves to delimit its scope.

`end` also marks the termination of a function, although in many cases it is optional. If your function contains one or more nested functions, then you must terminate every function in the file, whether nested or not, with `end`. This includes primary, nested, private, and local functions.

The `end` function also serves as the last index in an indexing expression. In that context, `end = (size(x,k))` when used as part of the `k`th index. Examples of this use are `X(3:end)` and `X(1,1:2:end-1)`. When using `end` to grow an array, as in `X(end+1)=5`, make sure `X` exists first.

You can overload the `end` statement for a user object by defining an `end` method for the object. The `end` method should have the calling sequence `end(obj,k,n)`, where `obj` is the user object, `k` is the index in the expression where the `end` syntax is used, and `n` is the total number of indices in the expression. For example, consider the expression

```
A(end-1,:)
```

The MATLAB software calls the `end` method defined for `A` using the syntax

```
end(A,1,2)
```

Examples

This example shows `end` used with the `for` and `if` statements.

```
for k = 1:n
 if a(k) == 0
  a(k) = a(k) + 2;
    end
end
```

In this example, `end` is used in an indexing expression.

```
A = magic(5)

A =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

B = A(end,2:end)

B =

    18    25     2     9
```

See Also

`break` | `for` | `if` | `return` | `switch` | `try` | `while` | `parfor`

EndInvoke

Purpose

Retrieve result of asynchronous call initiated by .NET System.Delegate BeginInvoke method

Syntax

```
result = EndInvoke(asyncResult)
[res0,...,resN] = EndInvoke(res0,...,resN,asyncResult)
```

Description

`result = EndInvoke(asyncResult)` retrieves result of asynchronous call initiated by `BeginInvoke` method.

`[res0,...,resN] = EndInvoke(res0,...,resN,asyncResult)` for methods with `out` and/or `ref` parameters.

Tips

- If the delegate contains `out` or `ref` parameters, the signature for the `EndInvoke` method follows the MATLAB mapping rules. For information, see “Call Delegates With out and ref Type Arguments”.

Input Arguments

asyncResult

.NET System.IAsyncResult object returned by `BeginInvoke`.

res0,...,resN

For methods with `out` and/or `ref` parameters, results of the asynchronous call. The number of arguments is the sum of:

- Number of return values (`0` or `1`).
- Number of `out` and `ref` arguments.

Output Arguments

result

Results of the asynchronous call.

res0,...,resN

For methods with `out` and/or `ref` parameters, results of the asynchronous call,

Examples

The following examples show how to call delegates with various input and output arguments. Each example contains:

1. The C# delegate signature. In order to execute the MATLAB code, you must build the delegate code into an assembly named `SignatureExamples` and load it into MATLAB. For information, see “Building a .NET Application for MATLAB Examples”.
2. An example MATLAB function to use with the delegate, which must exist on your path.
3. The `BeginInvoke` and `EndInvoke` signatures MATLAB creates. To display the signatures, create a delegate instance, `myDel`, and call the `methodsview` function.
4. Simple MATLAB example.

This examples shows how to use a delegate that has no return value.

1. C# delegate:

   ```
   public delegate void delint(Int32 arg);
   ```

2. MATLAB function to call:

   ```
   %Display input argument
   function dispfnc(A)
   %A = number
   ['Input is ' num2str(A)]
   end
   ```

3. MATLAB creates the following signatures. For `BeginInvoke`:

   ```
   System.IAsyncResult RetVal
       BeginInvoke (
           SignatureExamples.delint this,
           int32 scalar arg,
   ```

```
        System.AsyncCallback callback,
        System.Object object)
```

The `EndInvoke` signature:

```
    EndInvoke (
        SignatureExamples.delint this,
        System.IAsyncResult result)
```

4 Call `dispfnc`:

```
myDel = SignatureExamples.delint(@dispfnc);
asyncRes = myDel.BeginInvoke(6, [], []);
while asyncRes.IsCompleted ~= true
    pause(0.05); % Use pause() to let MATLAB process event
end
myDel.EndInvoke(asyncRes)
```

MATLAB displays:

```
Input is 6
```

This example shows how to use a delegate with a return value. The delegate does not have `out` or `ref` parameters.

1 C# delegate:

```
public delegate Int32 del2int(Int32 arg1, Int32 arg2);
```

2 MATLAB function to call:

```
%Add input arguments
function res = addfnc(A, B)
%A and B are numbers
res = A + B;
end
```

3 MATLAB creates the following signatures. For `BeginInvoke`:

```
System.IAsyncResult RetVal
    BeginInvoke (
        SignatureExamples.del2int this,
        int32 scalar arg1,
        int32 scalar arg2,
        System.AsyncCallback callback,
        System.Object object)
```

The `EndInvoke` signature:

```
int32 scalar RetVal
    EndInvoke (
        SignatureExamples.del2int this,
        System.IAsyncResult result)
```

4 Call `addfnc`:

```
myDel = SignatureExamples.del2int(@addfnc);
asyncRes = myDel.BeginInvoke(6,8,[],[]);
while asyncRes.IsCompleted ~= true
    pause(0.05); % Use pause() to let MATLAB process event
end
result = myDel.EndInvoke(asyncRes)
```

MATLAB displays:

```
result =
          14
```

This example shows how to use a delegate with a `ref` parameter, `refArg`, and no return value.

1 C# delegate:

```
public delegate void delrefvoid(ref Double refArg);
```

2 MATLAB maps the `ref` argument as both RHS and LHS arguments. MATLAB function to call:

```
%Increment input argument
function res = incfnc(A)
%A = number
res = A + 1;
end
```

3 MATLAB creates the following signatures. For `BeginInvoke`:

```
[System.IAsyncResult RetVal,
double scalar refArg]
    BeginInvoke (
        SignatureExamples.delrefvoid this,
        double scalar refArg,
        System.AsyncCallback callback,
        System.Object object)
```

The `EndInvoke` signature:

```
double scalar refArg
    EndInvoke (
        SignatureExamples.delrefvoid this,
        double scalar refArg,
        System.IAsyncResult result)
```

4 Call `incfnc`:

```
x = 6;
myDel = SignatureExamples.delrefvoid(@incfnc);
asyncRes = myDel.BeginInvoke(x,[],[]);
while asyncRes.IsCompleted ~= true
    pause(0.05); % Use pause() to let MATLAB process event
end
myRef = 0;
result = myDel.EndInvoke(myRef,asyncRes);
```

```
disp(['Increment of ' num2str(x) ' = ' num2str(result)]);
```

MATLAB displays:

```
Increment of 6 = 7
```

This example shows how to use a delegate with an `out` parameter, `argOut`, and one return value.

1 C# delegate:

```
public delegate Single deloutsingle(Single argIn, out Single argOut
```

2 MATLAB maps the `out` argument as a return value for a total of two return values. MATLAB function to call:

```
%Double input argument
function [res1 res2] = times2fnc(A)
res1 = A*2;
res2 = res1;
end
```

3 MATLAB creates the following signatures. For `BeginInvoke`:

```
[System.IAsyncResult RetVal,
single scalar argOut]
    BeginInvoke (
        SignatureExamples.deloutsingle this,
        single scalar argIn,
        System.AsyncCallback callback,
        System.Object object)
```

The `EndInvoke` signature:

```
[single scalar RetVal,
single scalar argOut]
    EndInvoke (
```

```
        SignatureExamples.deloutsingle this,
        System.IAsyncResult result)
```

4 Call `times2fnc`:

```
myDel = SignatureExamples.deloutsingle(@times2fnc);
asyncRes = myDel.BeginInvoke(6,[],[]);
while asyncRes.IsCompleted ~= true
    pause(0.05); % Use pause() to let MATLAB process event
end
[a1 a2] = myDel.EndInvoke(asyncRes);
a1
```

MATLAB displays:

```
a1 =
    12
```

See Also

`BeginInvoke`

How To

- "Calling .NET Methods Asynchronously"

Related Links

- MSDN Calling Synchronous Methods Asynchronously

Purpose

Last day of month

Syntax

```
E = eomday(Y, M)
```

Description

`E = eomday(Y, M)` returns the last day of the year and month given by corresponding elements of arrays `Y` and `M`.

Examples

Show the end of month for January through September for the year 1900:

```
eomday(1900, 1:9)
ans =
    31    28    31    30    31    30    31    31    30
```

Find the number of days during that period:

```
sum(eomday(1900, 1:9))
ans =
   273
```

Because 1996 is a leap year, the statement `eomday(1996,2)` returns `29`. To show all the leap years in the twentieth century, try:

```
y = 1900:1999;
E = eomday(y, 2);
y(find(E == 29))

ans =
  Columns 1 through 6
     1904      1908      1912      1916      1920      1924

  Columns 7 through 12
     1928      1932      1936      1940      1944      1948

  Columns 13 through 18
     1952      1956      1960      1964      1968      1972
```

```
Columns 19 through 24
   1976      1980      1984      1988      1992      1996
```

See Also

datenum | datevec | weekday

Purpose

Display class enumeration members and names

Syntax

```
enumeration ClassName
enumeration(obj)
m = enumeration(ClassName)
m = enumeration(obj)
[m,s] = enumeration(ClassName)
[m,s] = enumeration(obj)
```

Description

`enumeration ClassName` displays the names of the enumeration members for the MATLAB class with the name `ClassName`.

`enumeration(obj)` displays the names of the enumeration members for the class of `obj`.

`m = enumeration(ClassName)` returns the enumeration members for the class in the column vector `m` of objects.

`m = enumeration(obj)` returns the enumeration members for the class of object, `obj`, in the column vector `m` of objects.

`[m,s] = enumeration(ClassName)` returns the names of the enumeration members in the cell array of strings `s`. The names in `s` correspond element-wise to the enumeration members in `m`.

`[m,s] = enumeration(obj)` returns the enumeration members for the class of object, `obj`, in the column vector `m` of objects.

Tips

- An enumeration class that derives from a built-in class can specify more than one name for a given enumeration member.
- When you call the `enumeration` function with no output arguments, MATLAB displays only the first name for each enumeration member (as specified in the class definition). To see all available enumeration members and their names, use the two output arguments (for example, `[m, s] = enumeration(obj);`).

Input Arguments

ClassName

The name of the enumeration class, in single quotes.

obj

A instance of an enumeration class.

Output Arguments

m

Column vector of enumeration members.

s

Cell array of strings containing the enumeration names.

Examples

All examples use the following enumeration class.

```
classdef Boolean < logical
   enumeration
      No  (0)
      Yes (1)
      Off (0)
      On  (1)
   end
end
```

Display the names of the enumeration members for class `Boolean`:

```
enumeration Boolean
Enumerations for class Boolean:

    No
    Yes
```

Get the enumeration members for class `Boolean` in a column vector of objects:

```
members = enumeration('Boolean')
members =
```

```
    No
    Yes
```

Get all available enumeration members and their names:

```
[members, names] = enumeration('Boolean')
members =

    No
    Yes
    No
    Yes


names =

    'No'
    'Yes'
    'Off'
    'On'
```

See Also

`classdef`

Tutorials

- "Working with Enumerations"

eps

Purpose

Floating-point relative accuracy

Syntax

```
eps
d = eps(X)
eps('double')
eps('single')
```

Description

`eps` returns the distance from 1.0 to the next largest double-precision number, that is `eps = 2^(-52)`.

`d = eps(X)` is the positive distance from `abs(X)` to the next larger in magnitude floating point number of the same precision as `X`. `X` may be either double precision or single precision. For all `X`,

```
eps(X) = eps(-X) = eps(abs(X))
```

`eps('double')` is the same as `eps` or `eps(1.0)`.

`eps('single')` is the same as `eps(single(1.0))` or `single(2^-23)`.

Except for numbers whose absolute value is smaller than `realmin` , if `2^E <= abs(X) < 2^(E+1)`, then

```
eps(X) = 2^(E-23) if isa(X,'single')
eps(X) = 2^(E-52) if isa(X,'double')
```

For all `X` of class double such that `abs(X) <= realmin`, `eps(X) = 2^(-1074)`. Similarly, for all `X` of class single such that `abs(X) <= realmin('single')`, `eps(X) = 2^(-149)`.

Replace expressions of the form:

```
if Y < eps * ABS(X)
```

with

```
if Y < eps(X)
```

Examples

```
double precision
eps(1/2) = 2^(-53)
```

```
eps(1) = 2^(-52)
eps(2) = 2^(-51)
eps(realmax) = 2^971
eps(0) = 2^(-1074)

if(abs(x)) <= realmin, eps(x) = 2^(-1074)
eps(realmin/2) = 2^(-1074)
eps(realmin/16) = 2^(-1074)
eps(Inf) = NaN
eps(NaN) = NaN

single precision
eps(single(1/2)) = 2^(-24)
eps(single(1)) = 2^(-23)
eps(single(2)) = 2^(-22)
eps(realmax('single')) = 2^104
eps(single(0)) = 2^(-149)
eps(realmin('single')/2) = 2^(-149)
eps(realmin('single')/16) = 2^(-149)
if(abs(x)) <= realmin('single'), eps(x) = 2^(-149)
eps(single(Inf)) = single(NaN)
eps(single(NaN)) = single(NaN)
```

See Also

realmax | realmin | intmax

Purpose

Test for equality

Syntax

```
A == B
eq(A, B)
```

Description

`A == B` compares each element of array `A` for equality with the corresponding element of array `B`, and returns an array with elements set to logical 1 (`true`) where `A` and `B` are equal, or logical 0 (`false`) where they are not equal. Each input of the expression can be an array or a scalar value.

If both `A` and `B` are scalar (i.e., 1-by-1 matrices), then the MATLAB software returns a scalar value.

If both `A` and `B` are nonscalar arrays, then these arrays must have the same dimensions, and MATLAB returns an array of the same dimensions as `A` and `B`.

If one input is scalar and the other a nonscalar array, then the scalar input is treated as if it were an array having the same dimensions as the nonscalar input array. In other words, if input `A` is the number 100, and `B` is a 3-by-5 matrix, then `A` is treated as if it were a 3-by-5 matrix of elements, each set to 100. MATLAB returns an array of the same dimensions as the nonscalar input array.

When comparing handle objects, use `eq` or the == operator to test whether objects are the same handle. Use `isequal` to test if objects have equal property values, even if those objects are different handles.

`eq(A, B)` is called for the syntax `A == B` when either `A` or `B` is an object.

Examples

Create two 6-by-6 matrices, `A` and `B`, and locate those elements of `A` that are equal to the corresponding elements of `B`:

```
A = magic(6);
B = repmat(magic(3), 2, 2);

A == B
ans =
```

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

See Also

`ne` | `le` | `ge` | `lt` | `gt` | relational operators

eq (MException)

Purpose

Compare scalar `MException` objects for equality

Syntax

```
eObj1 == eObj2
```

Description

`eObj1 == eObj2` tests scalar `MException` objects `eObj1` and `eObj2` for equality, returning logical 1 (`true`) if the two objects are identical, otherwise returning logical 0 (`false`).

See Also

`try` | `catch` | `error` | `assert` | `MException` | `isequal(MException)` | `ne(MException)` | `getReport(MException)` | `disp(MException)` | `throw(MException)` | `rethrow(MException)` | `throwAsCaller(MException)` | `addCause(MException)` | | `last(MException)`

Purpose

Error function

Syntax

```
Y = erf(X)
```

Definitions

The error function `erf(X)` is twice the integral of the Gaussian distribution with 0 mean and variance of 1/2.

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

Description

`Y = erf(X)` returns the value of the error function for each element of real array `X`.

See Also

`erfc` | `erfcinv` | `erfcx` | `erfinv`

Purpose Complementary error function

Syntax

```
Y = erfc(X)
```

Definitions The complementary error function `erfc(X)` is defined as

$$\begin{aligned}\operatorname{erfc}(x) &= \frac{2}{\sqrt{\pi}}\int_x^{\infty} e^{-t^2}\,dt \\ &= 1 - \operatorname{erf}(x)\end{aligned}$$

Description `Y = erfc(X)` computes the value of the complementary error function.

Tips The relationship between the complementary error function `erfc` and the standard normal probability distribution returned by the Statistics Toolbox function `normcdf` is

$$\operatorname{normcdf}(x) = \left(\frac{1}{2}\right) \times \operatorname{erfc}\left(\frac{-x}{\sqrt{2}}\right)$$

See Also `erf` | `erfcinv` | `erfcx` | `erfinv`

Purpose

Inverse complementary error function

Syntax

```
X = erfcinv(Y)
```

Description

`X = erfcinv(Y)` returns the value of the inverse of the complementary error function for each element of `Y`. Elements of `Y` must be in the interval `[0 2]`. The function `erfcinv` satisfies $y = \mathrm{erfc}(x)$ for $2 \geq y \geq 0$ and $-\infty \geq x \geq \infty$.

Tips

The relationship between the inverse complementary error function `erfcinv` and the inverse standard normal probability distribution returned by the Statistics Toolbox function `norminv` is:

$$\mathrm{norminv}(p) = \left(-\sqrt{2}\right) \times \mathrm{erfcinv}(2p).$$

See Also

`erf` | `erfc` | `erfcx` | `erfinv`

Purpose

Scaled complementary error function

Syntax

```
Y = erfcx(X)
```

Definitions

The scaled complementary error function `erfcx(X)` is defined as

$$\mathrm{erfcx}(x) = e^{x^2}\,\mathrm{erfc}(x)$$

For large `X`, `erfcx(X)` is approximately $\left(\frac{1}{\sqrt{\pi}}\right)\frac{1}{x}$

Description

`Y = erfcx(X)` computes the value of the scaled complementary error function.

See Also

`erf` | `erfcinv` | `erfc` | `erfinv`

Purpose

Inverse error function

Syntax

```
X = erfinv(Y)
```

Description

`X = erfinv(Y)` returns the value of the inverse error function for each element of `Y`. Elements of `Y` should be in the interval `[-1 1]` or the function will return NaN. The function `erfinv` satisfies $y = \mathrm{erf}(x)$ for $-1 \le y \le 1$ and $-\infty \le x \le \infty$.

Examples

`erfinv(1)` is `Inf`

`erfinv(-1)` is `-Inf`.

See Also

`erf` | `erfcinv` | `erfcx` | `erfc`

error

Purpose

Display message and abort function

Syntax

```
error('msgIdent', 'msgString', v1, v2, ..., vN)
error('msgString', v1, v2, ...)
error('msgString')
error(msgStruct)
```

Description

`error('msgIdent', 'msgString', v1, v2, ..., vN)` generates an exception if the currently-running function tests for and confirms a faulty or unexpected condition. Depending on how the program has been designed to respond to the error, MATLAB either enters a `catch` block to handle the error condition, or exits the program.

The *`msgIdent`* argument is a unique *message identifier* string that MATLAB attaches to the error message when it throws the error. A message identifier has the format `component:mnemonic`. Its purpose is to better identify the source of the error (see Message Identifiers for more information).

The *`msgString`* argument is a character string that informs the user about the cause of the error and can also suggest how to correct the faulty condition. The *`msgString`* string can include escape sequences such as `\t` or `\n`, as well as any of the format specifiers supported by the `sprintf` function (such as `%s` or `%d`). Additional arguments *`v1`*, *`v2`*, `...`, *`vN`* provide values that correspond to and replace the conversion specifiers.

For example, if *`msgString`* is "Error on line `%d`, command `%s`", then *`v1`* is the line number at which the error was detected, and *`v2`* is the command that failed. See "Formatting Strings" for more detailed information on using string formatting commands.

All string input arguments must be enclosed in single quotation marks. If *`msgString`* is an empty string, the `error` command has no effect.

`error('msgString', v1, v2, ...)`reports an error without including a message identifier in the error report. Although including a message identifier in an error report is recommended, it is not required.

`error('msgString')` is the same as the above syntax, except that the *msgString* string contains no conversion specifiers, no escape sequences, and no substitution value (*v1*, *v2*, ...) arguments. All characters in *msgString* are interpreted exactly as they appear in the *msgString* argument. MATLAB displays the `\t` in `'C:\testFolder'` for example, as a backslash character followed by the letter `t`, and not as a horizontal tab.

`error(msgStruct)` accepts a scalar error structure input *msgStruct* with at least one of the fields `message`, `identifier`, and `stack`. When the *msgStruct* input includes a `stack` field, the `stack` field of the error will be set according to the contents of the `stack` input. When specifying a `stack` input, use the absolute file name and the entire sequence of functions that nests the function in the stack frame. This is the same as the string returned by `dbstack('-completenames')`. If *msgStruct* is an empty structure, no action is taken and `error` returns without exiting the function.

Tips

The `error` function captures what information it can about the error that occurred and stores it in a data structure that is an object of the `MException` class. This *error record* contains the error message string, message identifier, the error stack, and optionally an array of other exception objects that are intended to provide information as to the cause of the exception. See "Capture Information About Exceptions" for more information on how to access and use an exception object.

You can access information in the exception object using the `catch` function as documented in the `catch` reference page. If your program terminates because of an exception and returns control to the Command Prompt, you can access the exception object using the `MException.last` command.

The `error` function also determines where the error occurred and provides this information in the `stack` field of the `MException` object. This field contains a structure array that has the same format as the output of the `dbstack` function. This stack points to the line where the `error` function was called.

The following table shows the MATLAB functions that can be useful for throwing an exception:

Function	Description
error	Throw exception with specified error message.
assert	Evaluate given expression and throw exception if false.
throw	Throw exception based on specified MException object.
throwAsCaller	Throw exception that appears to have been thrown by the calling function.
rethrow	Reissue previously caught exception.

Examples

Example 1 — Simple Error Message

Write a short function errtest1 that throws an error when called with an incorrect number of input arguments. Include a message identifier 'myApp:argChk' and error message:

```
function errtest1(x, y)
if nargin ~= 2
    error('myApp:argChk', 'Wrong number of input arguments')
end
```

Call the function with an incorrect number of inputs. The call to nargin, a function that checks the number of inputs, fails and the program calls error:

```
errtest1(pi)

Error using errtest1 (line 3)
Wrong number of input arguments
```

If you run this function from the Command Window, you can use the MException.last method to view the exception object:

```
err = MException.last
err =
  MException

  Properties:
    identifier: 'myApp:argChk'
       message: 'Wrong number of input arguments'
         cause: {}
         stack: [1x1 struct]
  Methods

err.stack
ans =
    file: 'c:\work\errtest1.m'
    name: 'errtest1'
    line: 3
```

Example 2 — Special Characters

MATLAB converts special characters (like `\n` and `%d`) in the error message string only when you specify more than one input argument with `error`. In the single-argument case shown below, `\n` is taken to mean `backslash-n`. It is not converted to a newline character:

```
error('In this case, the newline \n is not converted.')

In this case, the newline \n is not converted.
```

But, when more than one argument is specified, MATLAB does convert special characters. This holds true regardless of whether the additional argument supplies conversion values or is a message identifier:

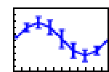
```
error('ErrorTests:convertTest', ...
      'In this case, the newline \n is converted.')

In this case, the newline
 is converted.
```

See Also

assert | try | dbstop | errordlg | warning | warndlg | MException | throw(MException) | rethrow(MException) | throwAsCaller(MException) | addCause(MException) | getReport(MException) | last(MException)

Purpose

Plot error bars along curve



Syntax

```
errorbar(Y,E)
errorbar(X,Y,E)
errorbar(X,Y,L,U)
errorbar(...,LineSpec)
h = errorbar(...)
```

Description

Error bars show the confidence intervals of data or the deviation along a curve.

`errorbar(Y,E)` plots `Y` and draws an error bar at each element of `Y`. The error bar is a distance of `E(i)` above and below the curve so that each bar is symmetric and `2*E(i)` long.

`errorbar(X,Y,E)` plots `Y` versus `X` with symmetric error bars `2*E(i)` long. `X`, `Y`, `E` must be the same size. When they are vectors, each error bar is a distance of `E(i)` above and below the point defined by `(X(i),Y(i))`. When they are matrices, each error bar is a distance of `E(i,j)` above and below the point defined by `(X(i,j),Y(i,j))`.

`errorbar(X,Y,L,U)` plots `X` versus `Y` with error bars `L(i)+U(i)` long specifying the lower and upper error bars. `X`, `Y`, `L`, and `U` must be the same size. When they are vectors, each error bar is a distance of `L(i)` below and `U(i)` above the point defined by `(X(i),Y(i))`. When they are matrices, each error bar is a distance of `L(i,j)` below and `U(i,j)` above the point defined by `(X(i,j),Y(i,j))`.

`errorbar(...,LineSpec)` uses the color and line style specified by the string `'LineSpec'`. The color is applied to the data line and error bars. The linestyle and marker are applied to the data line only. See `linespec` for examples of styles.

`h = errorbar(...)` returns handles to the errorbarseries objects created. `errorbar` creates one object for vector input arguments and

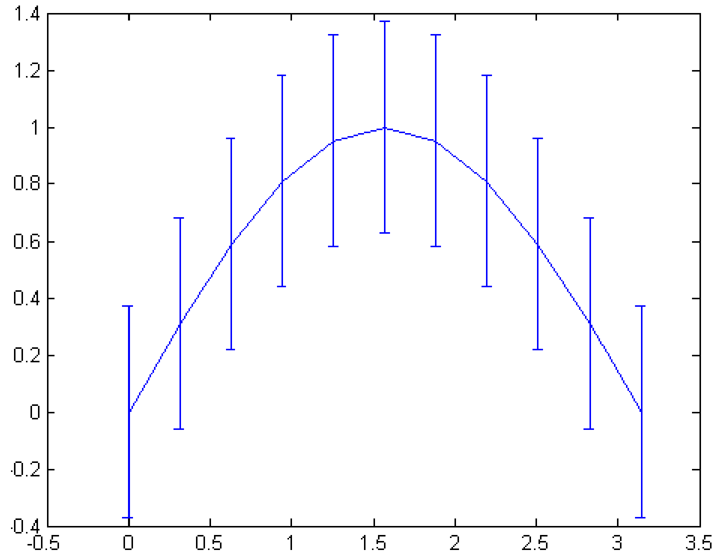
one object per column for matrix input arguments. See `errorbarseries properties` for more information.

When the arguments are all matrices, `errorbar` draws one line per matrix column. If `X` and `Y` are vectors, they specify one curve.

Examples

Draw symmetric error bars that are two standard deviation units in length:

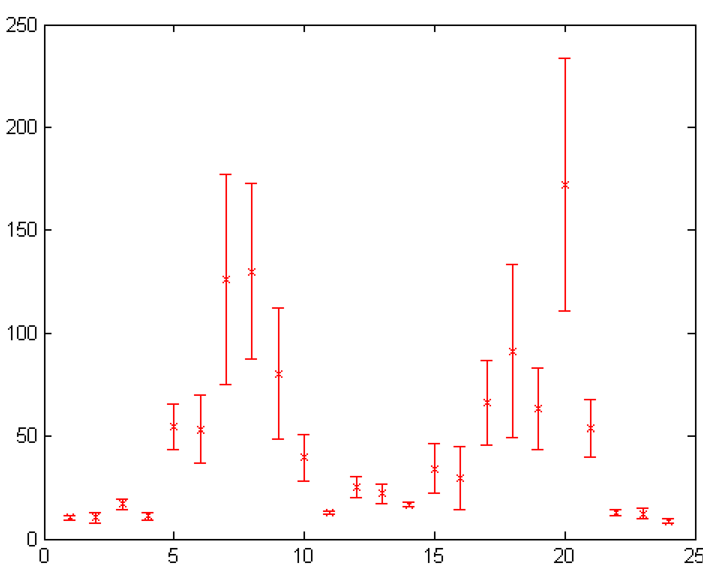
```
X = 0:pi/10:pi;
Y = sin(X);
E = std(Y)*ones(size(X));
errorbar(X,Y,E)
```



Plot the computed average traffic volume and computed standard deviations for three street locations over the course of a day using red 'x' markers:

```
load count.dat;
y = mean(count,2);
```

```
e = std(count,1,2);
figure
errorbar(y,e,'xr')
```



See Also

`corrcoef` | `linespec` | `plot` | `std` | `Errorbarseries Properties`

How To

- ConfidenceBounds

Errorbarseries Properties

Purpose

Description of errorbarseries properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands or the Property editor (`propertyeditor`).

Note that you cannot define default property values for errorbarseries objects. See "Plot Objects" for more information on errorbarseries objects.

Errorbarseries Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`Annotation`

`hg.Annotation` object (read-only)

Control the display of errorbarseries objects in legends. Specifies whether this errorbarseries object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the errorbarseries object is displayed in a figure legend.

IconDisplayStyle Value	Purpose
`on`	Include the errorbarseries object in a legend as one entry, but not its children objects
`off`	Do not include the errorbarseries or its children in a legend (default)
`children`	Include only the children of the errorbarseries as separate entries in the legend

Setting the IconDisplayStyle Property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the IconDisplayStyle Property

See "Controlling Legends" for more information and examples.

`BeingDeleted`

`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to

perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`

string | function handle

Button press callback function. Executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be:

- A string that is a valid MATLAB expression
- The name of a MATLAB file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`Children`
array of graphics object handles

Children of the errorbarseries object. An array containing the handles of all line objects parented to the errorbarseries object (whether visible or not).

If a child object's `HandleVisibility` property is `callback` or `off`, its handle does not show up in this object's `Children` property. If you want the handle in the `Children` property, set the root `ShowHiddenHandles` property to `on`. For example:

```
set(0,'ShowHiddenHandles','on')
```

`Clipping`
`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs appear outside the axes plot box. This occurs if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`Color`
ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object's color. The default value is `[0 0 0]` (`black`).

See the `ColorSpec` reference page for more information on specifying color. See "Adding Arrows and Lines to Graphs".

`CreateFcn`
string | function handle

Not available on errorbarseries objects.

`DeleteFcn`
function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback executed during object deletion. Executes when this object is deleted (for example, this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values. The default is an empty array.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`
string

String used by legend. The `legend` function uses the `DisplayName` property to label the errorbarseries object in the legend. The default is an empty string.

- If you specify string arguments with the `legend` function, MATLAB set `DisplayName` to the corresponding string and uses that string for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, MATLAB set `DisplayName` to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add a legend programmatically that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more information and examples.

`EraseMode`

`{normal} | none | xor | background`

Erase mode. Controls the technique MATLAB uses to draw and erase objects. Use alternative erase modes for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the object when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it because MATLAB stores no information about its former location.

- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object's color depends on the color of whatever is beneath it on the display.
- `background` — Erase the object by redrawing it in the axes background `Color`, or the figure background `Color` if the axes `Color` property is `none`. This damages objects that are behind the erased object, but properly colors the erased object.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors (for example, performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible.

- `callback` — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties. See also `findall`.

Handle Validity

Hidden handles are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`
`{on} | off`

Selectable by mouse click. Determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the curve or error bars that compose the graph. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`HitTestArea`
`on | {off}`

Select the object by clicking lines or area of extent. Select plot objects by:

- Clicking curve or error bars (default).
- Clicking anywhere in the extent of the plot.

When `HitTestArea` is `off`, you must click the curve or error bars to select the object. When `HitTestArea` is `on`, you can select this object by clicking anywhere within the extent of the plot (that is, anywhere within a rectangle that encloses all the lines).

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the `Interruptible` property affects only the callbacks for the`ButtonDownFcn` property. A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both the callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see "Control Callback Execution and Interruption".

`LData`
array equal in size to `XData` and `YData`

Errorbar length below data point. The `errorbar` function uses this data to determine the length of the errorbar below each data point. Specify these values in data units. See also `UData`.

`LDataSource`
string (MATLAB variable)

Link `LData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `LData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `LData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`. See the `refreshdata` reference page for more information.

`LineStyle`
`{-} | -- | : | -. | none`

Line style of errorbarseries object.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line
'none'	No line

Use `LineStyle` `none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`
size in points

Width of linear objects and edges of filled areas. Specify in points. 1 point = $^1/_{72}$ inch. The default is `0.5` points.

`Marker`
character (see table)

Marker symbol. Specifies marks that display at data points. You can set values for the `Marker` property independently from the `LineStyle` property. For a list of supported marker symbols, see the following table.

Marker Specifiers Table

Specifier	Marker Type
'+'	Plus sign
'o'	Circle

Specifier	Marker Type
’*’	Asterisk
’.’	Point
’x’	Cross
’square’ or ’s’	Square
’diamond’ or ’d’	Diamond
’^’	Upward-pointing triangle
’v’	Downward-pointing triangle
’>’	Right-pointing triangle
’<’	Left-pointing triangle
’pentagram’ or ’p’	Five-pointed star (pentagram)
’hexagram’ or ’h''’	Six-pointed star (hexagram)
’none’	No marker (default)

MarkerEdgeColor
    ColorSpec | none | {auto}

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `ColorSpec` — User-defined color.
- `none` — Specifies no color, which makes nonfilled markers invisible.
- `auto` — Uses same color as the `Color` property.

MarkerFaceColor
    ColorSpec | {none} | auto

Fill color for closed-shape markers. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `ColorSpec` — User-defined color.
- `none` — Makes the interior of the marker transparent, allowing the background to show through.
- `auto` — Sets the fill color to the axes `Color` property. If the axes `Color` property is `none`, sets the fill color to the figure `Color`.

`MarkerSize`
scalar

Marker size. Size of the marker in points. The default value is `6`.

Note MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`Parent`
handle of parent axes, hggroup, or hgtransform

Parent of object. Handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Object selection state. When you set this property to `on`, MATLAB displays selection handles at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this

property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing selection handles on the curve and error bars. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`Tag`
string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

For example, create an errorbarseries object and set the `Tag` property:

```
t = errorbar(Y,E,'Tag','errorbar1')
```

To access the errorbarseries object, use `findobj` to find the errorbarseries object's handle.

The following statement changes the `MarkerFaceColor` property of the object whose `Tag` is `errorbar1`.

```
set(findobj('Tag','errorbar1'),'MarkerFaceColor','red')
```

`Type`
string (read-only)

Type of graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For errorbarseries objects, `Type` is '`hggroup`'. The following statement finds all the hggroup objects in the current axes.

```
t = findobj(gca,'Type','hggroup');
```

`UData`

array equal in size to `XData` and `YData`

Errorbar length above data point. The `errorbar` function uses this data to determine the length of the errorbar above each data point. Specify these values in data units.

`UDataSource`

MATLAB variable, as a string

Link `UData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `UData`. The default value is an empty array.

```
set(h,'UDataSource','UDatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `UDataSource` does not change the object's `UData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

`UIContextMenu`

handle of uicontextmenu object

Associate a context menu with the errorbarseries object. Assign this property the handle of a uicontextmenu object created in the errorbarseries object's parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the

context menu whenever you right-click over the errorbarseries object.

`UserData`
array

User-specified data. Data you want to associate with the errorbarseries object (including cell arrays and structures). The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`
`{on} | off`

Visibility of errorbarseries object and its children. By default, errorbarseries object visibility is `on`. This means all children of the errorbarseries object are visible unless the child object's `Visible` property is `off`. Setting an errorbarseries object's `Visible` property to `off` also makes its children invisible.

`XData`
array

X-coordinates of the curve. The `errorbar` function plots a curve using the *x*-axis coordinates in the `XData` array. `XData` must be the same size as `YData`.

If you do not specify `XData` (which is the input argument `X`), the `errorbar` function uses the indices of `YData` to create the curve. See the `XDataMode` property for related information.

`XDataMode`
`{auto} | manual`

Use automatic or user-specified x-axis values. If you specify `XData` (by setting the `XData` property or specifying the input argument `x`), the `errorbar` function sets this property to `manual`.

If you set XDataMode to auto after having specified XData, the errorbar function resets the *x* tick-mark labels to the indices of the YData.

XDataSource
string (MATLAB variable)

Link XData to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the XData.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change XData.

You can use the refreshdata function to force an update of the object's data. refreshdata also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call refreshdata. See the refreshdata reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

YData
scalar | vector | matrix

Data defining curve. YData contains the data defining the curve. If YData is a matrix, the errorbar function displays a curve with error bars for each column in the matrix.

The input argument Y in the errorbar function calling syntax assigns values to YData.

`YDataSource`

string (MATLAB variable)

Link `YData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `YData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `YData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`. See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Purpose

Create and open error dialog box

Syntax

```
h = errordlg
h = errordlg(errorstring)
h = errordlg(errorstring,dlgname)
h = errordlg(errorstring,dlgname,createmode)
```

Description

`h = errordlg` creates and displays a dialog box with title `Error Dialog` that contains the string `This is the default error string.` The `errordlg` function returns the handle of the dialog box in `h`.

`h = errordlg(errorstring)` displays a dialog box with title `Error Dialog` that contains the string `errorstring`.

`h = errordlg(errorstring,dlgname)` displays a dialog box with title`dlgname` that contains the string `errorstring`.

`h = errordlg(errorstring,dlgname,`*`createmode`*`)` specifies whether the error dialog box is modal or nonmodal. Optionally, it can also specify an interpreter for `errorstring` and `dlgname`. The *`createmode`* argument can be a string or a structure.

If *`createmode`* is a string, it must be one of the values shown in the following table.

createmode Value	Description
`modal`	Replaces the error dialog box having the specified `Title`, that was last created or clicked on, with a modal error dialog box as specified. All other error dialog boxes with the same title are deleted. The dialog box which is replaced can be either modal or nonmodal.
`non-modal` (default)	Creates a new nonmodal error dialog box with the specified parameters. Existing

createmode Value	Description
	error dialog boxes with the same title are not deleted.
replace	Replaces the error dialog box having the specified Title, that was last created or clicked on, with a nonmodal error dialog box as specified. All other error dialog boxes with the same title are deleted. The dialog box which is replaced can be either modal or nonmodal.

Note A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution as well, use the uiwait function.

If you open a dialog with errordlg, msgbox, or warndlg using 'CreateMode','modal' and a non-modal dialog created with any of these functions is already present and *has the same name as the modal dialog*, the non-modal dialog closes when the modal one opens.

For more information about modal dialog boxes, see WindowStyle in the Figure Properties.

If CreateMode is a structure, it can have fields WindowStyle and Interpreter. WindowStyle must be one of the options shown in the table above. Interpreter is one of the strings 'tex' or 'none'. The default value for Interpreter is 'none'.

Tips

MATLAB sizes the dialog box to fit the string 'errorstring'. The error dialog box has an **OK** push button and remains on the screen until you press the **OK** button or the **Return** key. After pressing the button, the error dialog box disappears.

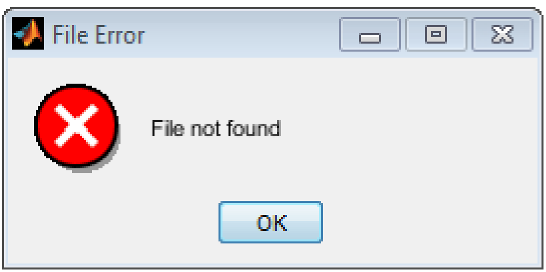
The appearance of the dialog box depends on the platform you use.

Examples

The function

```
errordlg('File not found','File Error');
```

displays this dialog box:



See Also

dialog | helpdlg | inputdlg | listdlg | msgbox | questdlg | warndlg | figure | uiwait | uiresume

etime

Purpose

Time elapsed between date vectors

Syntax

```
e = etime(t2,t1)
```

Description

`e = etime(t2,t1)` returns the number of seconds between two date vectors or matrices of date vectors, `t1` and `t2`.

Input Arguments

t2,t1 - Date vectors

1-by-6 vector | m-by-6 matrix

Date vectors, specified as 1-by-6 vectors or `m`-by-6 matrices containing `m` full date vectors in the format:

```
[Year Month Day Hour Minute Second]
```

Example: `[2012 03 27 11 50 01]`

Data Types

`double`

Examples

Compute Elapsed Time

Compute the time elapsed between a specific time and the current time, to 0.01-second accuracy.

Define the initial date and time and convert to date vector form.

```
format shortg
str = 'March 28, 2012 11:51:00';
t1 = datevec(str,'mmmm dd, yyyy HH:MM:SS')
```

```
t1 =
        2012           3          28          11          51           0
```

Determine the current date and time.

```
t2 = clock
```

```
t2 =
```

```
        2012           3          28          11          52
```

The `clock` function returns the current date and time as a date vector.

Use `etime` to compute the number of seconds between `t1` and `t2`.

```
e = etime(t2,t1)

e =

       62.357
```

Tips

- To time the duration of an event, use the `tic` and `toc` functions instead of `clock` and `etime`. The `clock` function is based on the system time, which can be adjusted periodically by the operating system, and thus might not be reliable in time comparison operations.

Algorithms

`etime` does not account for the following:

- Leap seconds.
- Daylight savings time adjustments.
- Differences in time zones.

See Also

`tic` | `toc` | `cputime` | `clock` | `now`

Purpose

Elimination tree

Syntax

```
p = etree(A)
p = etree(A,'col')
p = etree(A,'sym')
[p,q] = etree(...)
```

Description

`p = etree(A)` returns an elimination tree for the square symmetric matrix whose upper triangle is that of `A`. `p(j)` is the parent of column `j` in the tree, or `0` if `j` is a root.

`p = etree(A,'col')` returns the elimination tree of `A'*A`.

`p = etree(A,'sym')` is the same as `p = etree(A)`.

`[p,q] = etree(...)` also returns a postorder permutation `q` of the tree.

See Also

`treelayout` | `treeplot` | `etreeplot`

Purpose

Plot elimination tree

Syntax

```
etreeplot(A)
etreeplot(A,nodeSpec,edgeSpec)
```

Description

`etreeplot(A)` plots the elimination tree of `A` (or `A+A'`, if non-symmetric).

`etreeplot(A,nodeSpec,edgeSpec)` allows optional parameters `nodeSpec` and `edgeSpec` to set the node or edge color, marker, and linestyle. Use `''` to omit one or both.

See Also

`etree` | `treeplot` | `treelayout`

eval

Purpose

Execute MATLAB expression in text string

Syntax

```
eval(expression)
[output1,...,outputN] = eval(expression)
```

Description

`eval(expression)` evaluates the MATLAB code in the string `expression`. If you use `eval` within an anonymous function, nested function, or function that contains a nested function, the evaluated `expression` cannot create a variable.

`[output1,...,outputN] = eval(expression)` stores output from `expression` in the specified variables.

Tips

- Many common uses of the `eval` function are less efficient and are more difficult to read and debug than other MATLAB functions and language constructs. For more information, see "Alternatives to the eval Function".
- Whenever possible, do not include output arguments within the input to the `eval` function, such as `eval(['output = ',expression])`. The preferred syntax,

  ```
  output = eval(expression)
  ```

  allows the MATLAB parser to perform stricter checks on your code, preventing untrapped errors and other unexpected behavior.

Input Arguments

expression

String that contains a valid MATLAB expression.

To include a numeric value in the expression, convert it to a string with `int2str`, `num2str`, or `sprintf`.

Output Arguments

output1,...,outputN

Outputs from the evaluated `expression`.

Examples

Variable Name Evaluation

Select a matrix to plot at runtime.

This example requires that you have a matrix in the current workspace. For example:

```
aMatrix = magic(5);
```

Interactively request the name of a matrix to plot, and call `eval` to use its value.

```
expression = input('Enter the name of a matrix: ','s');
if (exist(expression,'var'))
    mesh(eval(expression))
end
```

If you type `aMatrix` at the input prompt, this code creates a mesh plot of `magic(5)`.

See Also

`assignin` | `evalc` | `evalin` | `feval` | `try`

Tutorials

- “Alternatives to the eval Function”
- “Variables in Nested and Anonymous Functions”

Purpose

Evaluate MATLAB expression with capture

Syntax

```
T = evalc(S)
[T, X, Y, Z, ...] = evalc(S)
```

Description

`T = evalc(S)` is the same as `eval(S)` except that anything that would normally be written to the command window, except for error messages, is captured and returned in the character array `T` (lines in `T` are separated by `\n` characters).

`[T, X, Y, Z, ...] = evalc(S)` is the same as `[X, Y, Z, ...] = eval(S)` except that any output is captured into `T`.

Tips

When you are using `evalc`, functions `diary`, `more`, and `input` are disabled.

See Also

`eval` | `evalin` | `assignin` | `feval` | `diary` | `input` | `more`

Purpose

Execute MATLAB expression in specified workspace

Syntax

```
evalin(ws, expression)
[a1, a2, a3, ...] = evalin(ws, expression)
```

Description

`evalin(ws, expression)` executes *`expression`*, a string containing any valid MATLAB expression, in the context of the workspace `ws`. `ws` can have a value of `'base'` or `'caller'` to denote the MATLAB base workspace or the workspace of the caller function. You can construct *`expression`* by concatenating substrings and variables inside square brackets:

```
expression = [string1, int2str(var), string2,...]
```

`[a1, a2, a3, ...] = evalin(ws, expression)` executes *`expression`* and returns the results in the specified output variables. Using the `evalin` output argument list is recommended over including the output arguments in the expression string:

```
evalin(ws,'[a1, a2, a3, ...] = function(var)')
```

The above syntax avoids strict checking by the MATLAB parser and can produce untrapped errors and other unexpected behavior.

Tips

The MATLAB base workspace is the workspace that is seen from the MATLAB command line (when not in the debugger). The caller workspace is the workspace of the function that called the currently running function. Note that the base and caller workspaces are equivalent in the context of a function that is invoked from the MATLAB command line.

`evalin('caller', expression)` finds only *variables* in the caller's workspace; it does not find *functions* in the caller. For this reason, you cannot use `evalin` to construct a handle to a function that is defined in the caller.

If you use `evalin('caller', expression)` in the MATLAB debugger after having changed your local workspace context with `dbup` or `dbdown`,

MATLAB evaluates the expression in the context of the function that is one level up in the stack from your current workspace context.

Examples

This example extracts the value of the variable `var` in the MATLAB base workspace and captures the value in the local variable `v`:

```
v = evalin('base', 'var');
```

Limitation

`evalin` cannot be used recursively to evaluate an expression. For example, a sequence of the form `evalin('caller', 'evalin(''caller'', ''x'')')` doesn't work.

See Also

`assignin` | `eval` | `evalc` | `feval` | `catch` | `try`

Purpose

Base class for all data objects passed to event listeners

Description

The `event` package contains the `event.EventData` class, which defines the data objects passed to event listeners. If you want to provide additional information to event listeners, you can do so by subclassing `event.EventData`. See "Defining Event-Specific Data" for more information.

Properties

The `event.EventData` class defines two properties and no methods:

- `EventName` — The name of the event described by this data object.
- `Source` — The source object whose class defines the event described by the data object.

See Also

`event.PropertyEvent`

How To

- "Learning to Use Events and Listeners"

event.listener

Purpose

Class defining listener objects

Syntax

`lh = event.listener(Hobj,'EventName',@CallbackFunction)`

Description

The `event.listener` class defines listener objects. Listener objects respond to the specified event and identify the callback function to invoke when the event is triggered.

`lh = event.listener(Hobj,'EventName',@CallbackFunction)` creates an `event.listener` object, `lh`, for the event named in *`EventName`*, on the specified object, `Hobj`.

If `Hobj` is an array of object handles, the listener responds to the named event on any of the objects referenced in the array.

The listener callback function must accept at least two input arguments. For example,

```
function CallbackFunction(source,eventData)
   ...
end
```

where `source` is the object that is the source of the event and `eventData` is an `event.EventData` object.

The `event.listener` class is a `handle` class.

Limiting Listener Lifecycle

Generally, you create a listener object using `addlistener`. However, you can call the `event.listener` constructor directly to create a listener. When you use the `event.listener` constructor, the listener's lifecycle is not tied to the object(s) being listened to—once the listener object goes out of scope, the listener no longer exists. See "Ways to Create Listeners" for more information on creating listener objects.

Removing a Listener

If you call `delete(lh)` on the listener object, the listener ceases to exist, which means the event no longer causes the listener callback function to execute.

Disabling a Listener

You can enable or disable a listener by setting the value of the listener's `Enabled` property (see Properties table below).

More Information on Events and Listeners

See "Events" for more information and examples of how to use events and listeners.

Properties

Property	Purpose
`Source`	Cell array of source objects
`EventName`	Name of the event
`Callback`	Function to execute when the event is triggered and the `Enabled` property is set to `true`
`Enabled`	The callback executes when the event occurs if and only if `Enabled` is set to `true` (the default).
`Recursive`	When `false` (the default), this listener does not execute recursively. Therefore, if the callback triggers its own event, the listener does not execute again. When `true`, the listener callback can cause the same event that triggered the callback. This scheme can lead to infinite recursion, which ends when the MATLAB recursion limit eventually triggers an error.

See Also

`addlistener` | `delete` | `event.proplistener`

event.PropertyEvent

Purpose

Data for property events

Description

The `event.PropertyEvent` class defines the data objects passed to listeners of the `meta.property` events:

- `PreGet`
- `PostGet`
- `PreSet`
- `PostSet`

`event.PropertyEvent` is a sealed subclass of `event.EventData` (i.e., you cannot subclass `event.PropertyEvent`).

Properties

`event.PropertyEvent` inherits the first two properties from the `event.EventData`, and defines one new property:

- `EventName` — One of the four event names listed in the Description section
- `Source` — `meta.property` object that triggers the event
- `AffectedObject` — The object whose property is affected.

See Also

`event.EventData` | `meta.property`

How To

- “Listen for Changes to Property Values”

Purpose

Define listener object for property events

Syntax

```
lh = event.proplistener(Hobj,Properties,'PropEvent',
   @CallbackFunction)
```

Description

`lh = event.proplistener(Hobj,Properties,'PropEvent',@CallbackFunction)` creates a property listener object for one or more properties on the specified object.

- `Hobj` — handle of object whose property or properties are to be listened to. If `Hobj` is an array, the listener responds to the named event on all objects in the array.
- `Properties` — an object array or a cell array of `meta.property` object handles representing the properties to which you want to listen.
- `PropEvent` — must be one of the strings: `PreSet`, `PostSet`, `PreGet`, `PostGet`
- `@CallbackFunction` — function handle to the callback function that executes when the event occurs.

The `event.proplistener` class defines property event listener objects. It is a subclass of the `event.listener` class and adds one property to those defined by `event.listener`:

- `Object` — Cell array of objects whose property events are being listened to.

You can call the `event.proplistener` constructor instead of calling `addlistener` to create a property listener. However, when you do not use `addlistener`, the listener's lifecycle is not tied to the object(s) being listened to.

See "Listen for Changes to Property Values".

See "Getting Information About Properties" for more information on using `meta.property` objects.

See Also

`event.listener` | `addlistener`

eventlisteners

Purpose

List event handler functions associated with COM object events

Syntax

```
info = h.eventlisteners
info = eventlisteners(h)
```

Description

`info = h.eventlisteners` lists the events and their event handler routines registered with COM object `h`. The function returns a cell array of strings `info`, with each row containing the name of a registered event and the handler routine for that event. If the object has no registered events, `eventlisteners` returns an empty cell array. You can register events either when you create the control, using `actxcontrol`, or at any time afterwards, using `registerevent`.

`info = eventlisteners(h)` is an alternate syntax.

COM functions are available on Microsoft Windows systems only.

Examples

Manage events for an instance of the MATLAB control `mwsamp`:

```
f = figure('position', [100 200 200 200]);
%Create an mwsamp control and
%register the Click event
h = actxcontrol('mwsamp.mwsampctrl.2', ...
    [0 0 200 200], f, ...
    {'Click' 'myclick'});
h.eventlisteners
```

MATLAB displays the event name and its event handler, `myclick`:

```
ans =
    'Click'    'myclick'
```

Register two more events, `DblClick` and `MouseDown`:

```
h.registerevent({'DblClick', 'my2click'; 'MouseDown' 'mymoused'});
h.eventlisteners
```

MATLAB displays all event names and handlers:

```
ans =
    'Click'         'myclick'
    'Dblclick'      'my2click'
    'Mousedown'     'mymoused'
```

Unregister all events for the control:

```
h.unregisterallevents
h.eventlisteners
```

MATLAB displays an empty cell array, indicating the control has no registered events:

```
ans =
     {}
```

See Also

events (COM) | registerevent | unregisterevent | unregisterallevents | isevent | actxcontrol

events

Purpose

Event names

Syntax

```
events('classname')
events(obj)
e = events(...)
```

Description

`events('classname')` displays the names of the public events for the MATLAB class *`classname`*, including events inherited from superclasses.

`events(obj)` `obj` is a scalar or array of objects of a MATLAB class.

`e = events(...)` returns the event names in a cell array of strings.

An event is public when the value of its `ListenAccess` attribute is `public` and its `Hidden` attribute value is `false` (default values for both attributes). See "Event Attributes" for a complete list of attributes.

`events` is also a MATLAB class-definition keyword. See `classdef` for more information on class definition keywords.

Examples

Get the names of the public events for the `handle` class:

```
events('handle')
Events for class handle:

    ObjectBeingDestroyed
```

See Also

`properties` | `methods`

Tutorials

- "Events"

Purpose

List of events COM object can trigger

Syntax

```
S = h.events
S = events(h)
```

Description

`S = h.events` returns structure array `S` containing all events, both registered and unregistered, known to the COM object, and the function prototype used when calling the event handler routine. For each array element, the structure field is the event name and the contents of that field is the function prototype for that event's handler.

`S = events(h)` is an alternate syntax.

Tips

COM functions are available on Microsoft Windows systems only.

Examples

List Control Events Example

Create an `mwsamp` control and list all events:

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.2', [0 0 200 200], f);
h.events
```

MATLAB software displays information similar to:

```
Click = void Click()
DblClick = void DblClick()
MouseDown = void MouseDown(int16 Button, int16 Shift,
  Variant x, Variant y)
Event_Args = void Event_Args(int16 typeshort, int32 typelong,
  double typedouble, string typestring, bool typebool)
```

Assign the output to a variable and get one field of the returned structure:

```
ev = h.events;
ev.MouseDown
```

MATLAB displays:

```
ans =
  void MouseDown(int16 Button, int16 Shift, Variant x, Variant y)
```

List Workbook Events Example

Open a Microsoft Excel application and list all events for a `Workbook` object:

```
myApp = actxserver('Excel.Application');
wbs = myApp.Workbooks;
wb = wbs.Add;
wb.events
```

The MATLAB software displays all events supported by the `Workbook` object.

```
 Open = void Open()
 Activate = void Activate()
 Deactivate = void Deactivate()
 BeforeClose = void BeforeClose(bool Cancel)
                 .
                 .
```

See Also

`isevent` | `eventlisteners` | `registerevent` | `unregisterevent` | `unregisterallevents`

Purpose

Execute MATLAB command in Automation server

Syntax

MATLAB Client

```
result = h.Execute('command')
result = Execute(h, 'command')
result = invoke(h, 'Execute', 'command')
```

IDL Method Signature

```
BSTR Execute([in] BSTR command)
```

Microsoft Visual Basic® Client

```
Execute(command As String) As String
```

Description

The `Execute` function executes the MATLAB statement specified by the string `command` in the MATLAB Automation server attached to handle `h`.

The server returns output from the command in the string, `result`. The `result` string also contains any warning or error messages that might have been issued by MATLAB software as a result of the command.

Note that if you terminate the MATLAB command string with a semicolon and there are no warnings or error messages, `result` might be returned empty.

Tips

If you want to be able to display output from `Execute` in the client window, you must specify an output variable (i.e., `result` in the above syntax statements).

Server function names, like `Execute`, are case sensitive when used with dot notation (the first syntax shown).

All three versions of the MATLAB client syntax perform the same operation.

If there is an error, the `Execute` function returns the MATLAB error message with the characters `???` prepended to the text.

COM functions are available on Microsoft Windows systems only.

Examples

Execute the MATLAB `version` function in the server and return the output to the MATLAB client.

MATLAB Client

```
h = actxserver('matlab.application');
server_version = h.Execute('version')
server_version =
ans =
   6.5.0.180913a (R13)
```

Visual Basic .NET Client

```
Dim Matlab As Object
Dim server_version As String
Matlab = CreateObject("matlab.application")
server_version = Matlab.Execute("version")
```

See Also

`Feval` | `PutFullMatrix` | `GetFullMatrix` | `PutCharArray` | `GetCharArray`

exifread

Purpose

Read EXIF information from JPEG and TIFF image files

Syntax

Note `exifread` will be removed in a future release. Use `imfinfo` instead.

```
output = exifread(filename)
```

Description

`output = exifread(filename)` reads the Exchangeable Image File Format (EXIF) data from the file specified by the string `filename`. `filename` must specify a JPEG or TIFF image file. `output` is a structure containing metadata values about the image or images in `imagefile`.

Note `exifread` returns all EXIF tags and does not process them in any way.

EXIF is a standard used by digital camera manufacturers to store information in the image file, such as, the make and model of a camera, the time the picture was taken and digitized, the resolution of the image, exposure time, and focal length. For more information about EXIF and the meaning of metadata attributes, see `http://www.exif.org/`.

See Also

`imfinfo` | `imread`

exist

Purpose

Check existence of variable, function, folder, or class

Syntax

```
exist name
exist name kind
A = exist('name','kind')
```

Description

`exist name` returns the status of `name`:

0	`name` does not exist.
1	`name` is a variable in the workspace.
2	One of the following is true: • `name` exists on your MATLAB search path as a file with extension `.m`. • `name` is the name of an ordinary file on your MATLAB search path. • `name` is the full pathname to any file.
3	`name` exists as a MEX- or DLL-file on your MATLAB search path.
4	`name` exists as a Simulink model or library file on your MATLAB search path.
5	`name` is a built-in MATLAB function.
6	`name` is a P-file on your MATLAB search path.
7	`name` is a folder.
8	`name` is a class. (`exist` returns 0 for Java classes if you start MATLAB with the `-nojvm` option.)

If `name` is a class, then `exist('name')` returns an `8`. However, if `name` is a class file, then `exist('name')` returns a `2`.

If a file or folder is not on the search path, then `name` must specify either a full pathname, a partial pathname relative to `MATLABPATH`, a partial

pathname relative to your current folder, or the file or folder must reside in your current working folder.

If `name` specifies a filename, that filename may include an extension to preclude conflicting with other similar filenames. For example, `exist('file.ext')`.

`exist name` *`kind`* returns the status of `name` for the specified *`kind`*. If `name` of type *`kind`* does not exist, it returns `0`. The *`kind`* argument may be one of the following:

`builtin`	Checks only for built-in functions.
`class`	Checks only for classes.
`dir`	Checks only for folders.
`file`	Checks only for files or folders.
`var`	Checks only for variables.

If you do not specify a *`kind`* argument, and `name` belongs to more than one of these categories, `exist` returns one value according to the order of evaluation shown in the table below. For example, if `name` matches both a folder and a file that defines a MATLAB function, `exist` returns `7`, identifying it as a folder.

Order of Evaluation	Return Value	Type of Entity
1	1	Variable
2	5	Built-in
3	7	Folder
4	3	MEX or DLL-file
5	4	MDL-file
6	6	P-file

Order of Evaluation	Return Value	Type of Entity
7	2	MATLAB function
8	8	Class

`A = exist('name','kind')` is the function form of the syntax.

Tips

If `name` specifies a filename, MATLAB attempts to locate the file, examines the filename extension, and determines the value to return based on the extension alone. MATLAB does not examine the contents or internal structure of the file.

You can specify a partial path to a folder or file. A partial pathname is a pathname relative to the MATLAB path that contains only the trailing one or more components of the full pathname. For example, both of the following commands return `2`, identifying `mkdir.m` as a MATLAB function. The first uses a partial pathname:

```
exist('matlab/general/mkdir.m')
exist([matlabroot '/toolbox/matlab/general/mkdir.m'])
```

To check for the existence of more than one variable, use the `ismember` function. For example,

```
a = 5.83;
c = 'teststring';
ismember({'a','b','c'},who)

ans =

     1     0     1
```

Examples

This example uses `exist` to check whether a MATLAB function is a built-in function or a file:

```
type = exist('plot')
type =
```

```
     5
```

This indicates that `plot` is a built-in function.

Run `exist` on a class folder and then on the constructor within that folder:

```
exist('@portfolio')
ans =
     7               % @portfolio is a folder

exist('@portfolio\portfolio')
ans =
     2               % portfolio is a MATLAB  function
```

The following example indicates that `testresults` is both a variable in the workspace and a folder on the search path:

```
exist('testresults','var')
ans =
     1
exist('testresults','dir')
ans =
     7
```

See Also

`assignin` | `computer` | `dir` | `evalin` | `help` | `inmem` | `isfield` | `isempty` | `lookfor` | `mfilename` | `what` | `which` | `who`

exit

Purpose

Terminate MATLAB program (same as `quit`)

Alternatives

As an alternative to the `exit` function, click the Close box in the MATLAB desktop.

Syntax

```
exit
exit(code)
```

Description

`exit` terminates the current session of MATLAB after running `finish.m`, if the file `finish.m` exists. It performs the same as `quit` and takes the same termination options, such as **`force`**.

`exit(code)` returns exit code when calling a MATLAB command from the system command line.

See Also

`quit` | `finish`

Purpose

Exponential

Syntax

```
Y = exp(X)
```

Description

`Y = exp(X)` returns the exponential for each element of `X`. `exp` operates element-wise on arrays. For complex $x + i * y$, `exp` returns the complex exponential $e^z = e^x(\cos y + i \sin y)$. Use `expm` for matrix exponentials.

Examples

Find the value of $e^{i\pi}$:

```
y=exp(i*pi)
```

returns

```
y =

  -1.0000 + 0.0000i
```

See Also

`expm` | `log`

expint

Purpose

Exponential integral

Syntax

```
Y = expint(X)
```

Definitions

The exponential integral computed by this function is defined as

$$E_1(x) = \int_x^{\infty} e^{-t} / t \, dt$$

Another common definition of the exponential integral function is the Cauchy principal value integral

$$\mathrm{Ei}(x) = \int_{-\infty}^{x} e^{t} / t \, dt$$

which, for real positive x, is related to `expint` as

$$E_1(-x) = -\mathrm{Ei}(x) - i\pi$$

Description

`Y = expint(X)` evaluates the exponential integral for each element of `X`.

References

[1] Abramowitz, M. and I. A. Stegun. *Handbook of Mathematical Functions*. Chapter 5, New York: Dover Publications, 1965.

Purpose

Matrix exponential

Syntax

```
Y = expm(X)
```

Description

`Y = expm(X)` computes the matrix exponential of `X`.

Although it is not computed this way, if `X` has a full set of eigenvectors `V` with corresponding eigenvalues `D`, then

```
[V,D] = EIG(X) and EXPM(X) = V*diag(exp(diag(D)))/V
```

Use `exp` for the element-by-element exponential.

Algorithms

`expm` uses the Padé approximation with scaling and squaring. See reference [3], below.

Note The files, `expmdemo1.m`, `expmdemo2.m`, and `expmdemo3.m` illustrate the use of Padé approximation, Taylor series approximation, and eigenvalues and eigenvectors, respectively, to compute the matrix exponential. References [1] and [2] describe and compare many algorithms for computing a matrix exponential.

Examples

This example computes and compares the matrix exponential of `A` and the exponential of `A`.

```
A = [1         1         0
     0         0         2
     0         0        -1 ];

expm(A)
ans =
    2.7183   1.7183        1.0862
    0        1.0000        1.2642
    0             0        0.3679
```

```
exp(A)
ans =
    2.7183    2.7183    1.0000
    1.0000    1.0000    7.3891
    1.0000    1.0000    0.3679
```

Notice that the diagonal elements of the two results are equal. This would be true for any triangular matrix. But the off-diagonal elements, including those below the diagonal, are different.

References

[1] Golub, G. H. and C. F. Van Loan, *Matrix Computation*, p. 384, Johns Hopkins University Press, 1983.

[2] Moler, C. B. and C. F. Van Loan, “Nineteen Dubious Ways to Compute the Exponential of a Matrix,” *SIAM Review 20*, 1978, pp. 801–836. Reprinted and updated as “Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later,” *SIAM Review 45*, 2003, pp. 3–49.

[3] Higham, N. J., “The Scaling and Squaring Method for the Matrix Exponential Revisited,” *SIAM J. Matrix Anal. Appl.*, 26(4) (2005), pp. 1179–1193.

See Also

`exp` | `expm1` | `funm` | `logm` | `eig` | `sqrtm`

Purpose

Compute exp(x)-1 accurately for small values of x

Syntax

```
y = expm1(x)
```

Description

`y = expm1(x)` computes `exp(x)-1`, compensating for the roundoff in `exp(x)`.

For small `x`, `expm1(x)` is approximately `x`, whereas `exp(x)-1` can be zero.

See Also

`exp` | `expm` | `log1p`

export2wsdlg

Purpose

Export variables to workspace

Syntax

```
export2wsdlg(checkboxlabels,defaultvariablenames,
itemstoexport)
export2wsdlg(checkboxlabels,defaultvariablenames,
itemstoexport,title)
export2wsdlg(checkboxlabels,defaultvariablenames,
itemstoexport,title,selected)
export2wsdlg(checkboxlabels,defaultvariablenames,
itemstoexport,title,selected,helpfunction)
export2wsdlg(checkboxlabels,defaultvariablenames,
itemstoexport,title,selected,helpfunction,functionlist)
hdialog = export2wsdlg(...)
[hdialog,ok_pressed] = export2wsdlg(...)
```

Description

`export2wsdlg(checkboxlabels,defaultvariablenames, itemstoexport)` creates a dialog with a series of check boxes and edit fields. `checkboxlabels` is a cell array of labels for the check boxes. `defaultvariablenames` is a cell array of strings that serve as a basis for variable names that appear in the edit fields. `itemstoexport` is a cell array of the values to be stored in the variables. If there is only one item to export, `export2wsdlg` creates a text control instead of a check box.

Note By default, the dialog box is modal. A modal dialog box prevents the user from interacting with other windows before responding.

`export2wsdlg(checkboxlabels,defaultvariablenames, itemstoexport,title)` creates the dialog with `title` as its title.

`export2wsdlg(checkboxlabels,defaultvariablenames, itemstoexport,title,selected)` creates the dialog allowing the user to control which check boxes are checked. `selected` is a logical array whose length is the same as `checkboxlabels`. True indicates that the check box should initially be checked, false unchecked.

`export2wsdlg(checkboxlabels,defaultvariablenames, itemstoexport,title,selected,helpfunction)` creates the dialog with a help button. `helpfunction` is a callback that displays help.

`export2wsdlg(checkboxlabels,defaultvariablenames, itemstoexport,title,selected,helpfunction,functionlist)` creates a dialog that enables the user to pass in `functionlist`, a cell array of functions and optional arguments that calculate, then return the value to export. `functionlist` should be the same length as `checkboxlabels`.

`hdialog = export2wsdlg(...)` returns the handle of the dialog.

`[hdialog,ok_pressed] = export2wsdlg(...)` sets `ok_pressed` to true if the OK button is pressed, or false otherwise. If two return arguments are requested, `hdialog` is `[]` and the function does not return until the dialog is closed.

The user can edit the text fields to modify the default variable names. If the same name appears in multiple edit fields, `export2wsdlg` creates a structure using that name. It then uses the `defaultvariablenames` as fieldnames for that structure.

The lengths of `checkboxlabels`, `defaultvariablenames`, `itemstoexport` and `selected` must all be equal.

The strings in `defaultvariablenames` must be unique.

Examples

This example creates a dialog box that enables the user to save the variables `sumA` and/or `meanA` to the workspace. The dialog box title is Save Sums to Workspace.

```
A = randn(10,1);
checkLabels = {'Save sum of A to variable named:' ...
               'Save mean of A to variable named:'};
varNames = {'sumA','meanA'};
items = {sum(A),mean(A)};
export2wsdlg(checkLabels,varNames,items,...
             'Save Sums to Workspace');
```

Purpose

Identity matrix

Syntax

```
I = eye
I = eye(n)
I = eye(n,m)
I = eye(sz)

I = eye(classname)
I = eye(n,classname)
I = eye(n,m,classname)
I = eye(sz,classname)

I = eye('like',p)
I = eye(n,'like',p)
I = eye(n,m,'like',p)
I = eye(sz,'like',p)
```

Description

`I = eye` returns the scalar, `1`.

`I = eye(n)` returns an `n`-by-`n` identity matrix with ones on the main diagonal and zeros elsewhere.

`I = eye(n,m)` returns an `n`-by-`m` matrix with ones on the main diagonal and zeros elsewhere.

`I = eye(sz)` returns an array with ones on the main diagonal and zeros elsewhere. The size vector, `sz`, defines `size(I)`. For example, `eye([2,3])` returns a 2-by-3 array with ones on the main diagonal and zeros elsewhere.

`I = eye(classname)` returns a scalar, `1`, where the string, `classname`, specifies the data type. For example, `eye('int8')` returns a scalar, 8-bit integer.

`I = eye(n,classname)` returns an `n`-by-`n` identity matrix of data type `classname`.

`I = eye(n,m,classname)` returns an `n`-by-`m` matrix of data type `classname` with ones on the main diagonal and zeros elsewhere.

`I = eye(sz,classname)` returns a matrix with ones on the main diagonal and zeros elsewhere. The size vector, `sz`, defines `size(I)` and `classname` defines `class(I)`.

`I = eye('like',p)` returns a scalar, `1`, with the same data type, sparsity, and complexity (real or complex) as the numeric variable, `p`.

`I = eye(n,'like',p)` returns an `n`-by-`n` identity matrix like `p`.

`I = eye(n,m,'like',p)` returns an `n`-by-`m` matrix like `p`.

`I = eye(sz,'like',p)` returns a matrix like `p` where the size vector, `sz`, defines `size(I)`.

Input Arguments

n - Size of first dimension of `I`

integer value

Size of first dimension of `I`, specified as an integer value.

- If `n` is the only integer input argument, then `I` is a square n-by-n identity matrix.
- If `n` is `0`, then `I` is an empty matrix.
- If `n` is negative, then it is treated as `0`.

Data Types

`double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

m - Size of second dimension of I

integer value

Size of second dimension of I, specified as an integer value.

- If m is 0, then I is an empty matrix.
- If m is negative, then it is treated as 0.

Data Types

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

sz - Size of I

row vector of no more than two integer values

Size of I, specified as a row vector of no more than two integer values.

- If an element of sz is 0, then I is an empty matrix.
- If an element of sz is negative, then the element is treated as 0.

Example: sz = [2,3] defines I as a 2-by-3 matrix.

Data Types

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

classname - Output class

'double' (default) | 'single' | 'int8' | 'uint8'

Output class, specified as 'double', 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64', or 'uint64'.

Data Types

char

p - Prototype

numeric variable

Prototype, specified as a numeric variable.

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

Examples

Square Identity Matrix

Create a 4-by-4 identity matrix.

```
I = eye(4)

I =

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

Rectangular Matrix

Create a 2-by-3 identity matrix.

```
I = eye(2,3)

I =

     1     0     0
     0     1     0
```

Identity Vector

Create a 3-by-1 identity vector.

```
sz = [3,1];
I = eye(sz)

I =

     1
     0
```

```
     0
```

Nondefault Numeric Data Type

Create a 3-by-3 identity matrix whose elements are 32-bit unsigned integers.

```
I = eye(3,'uint32'),
class(I)
```

```
I =

           1           0           0
           0           1           0
           0           0           1


ans =

uint32
```

Complex Identity Matrix

Create a 2-by-2 identity matrix that is not real valued, but instead is complex like an existing array.

Define a complex vector.

```
p = [1+2i 3i];
```

Create an identity matrix that is complex like `p`.

```
I = eye(2,'like',p)
```

```
I =

   1.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   1.0000 + 0.0000i
```

Sparse Identity Matrix

Define a 5-by-5 sparse matrix.

```
p = sparse(5,5,pi);
```

Create a 5-by-5 identity matrix that is sparse like P.

```
I = eye(5,'like',p)
```

```
I =

   (1,1)        1
   (2,2)        1
   (3,3)        1
   (4,4)        1
   (5,5)        1
```

Size and Numeric Data Type Defined by Existing Array

Define a 2-by-2 matrix of single precision.

```
p = single([1 3 ; 2 4]);
```

Create an identity matrix that is the same size and data type as P.

```
I = eye(size(p),'like',p),
class(I)
```

```
I =

     1     0
     0     1


ans =

single
```

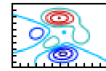
See Also

`speye` | `ones` | `zeros`

Concepts

Purpose

Easy-to-use contour plotter



Syntax

```
ezcontour(fun)
ezcontour(fun,domain)
ezcontour(...,n)
ezcontour(axes_handle,...)
h = ezcontour(...)
```

Description

`ezcontour(fun)` plots the contour lines of `fun(x,y)` using the `contour` function. `fun` is plotted over the default domain: $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

`fun` can be a function handle for a MATLAB file function or an anonymous function (see `function handle` and "Anonymous Functions") or a string (see Tips).

`ezcontour(fun,domain)` plots `fun(x,y)` over the specified `domain`. `domain` can be either a 4-by-1 vector `[xmin, xmax, ymin, ymax]` or a 2-by-1 vector `[min, max]` (where `min` < x < `max`, `min` < y < `max`).

`ezcontour(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezcontour(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezcontour(...)` returns the handles to contour objects in `h`.

`ezcontour` automatically adds a title and axis labels.

Tips

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the string expression you pass to `ezcontour`. For example, the MATLAB syntax for a contour plot of the expression

```
sqrt(x.^2 + y.^2)
```

is written as

```
ezcontour('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezcontour`.

If the function to be plotted is a function of the variables u and v (rather than x and y), the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezcontour('u^2 - v^3',[0,1],[3,6])` plots the contour lines for $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezcontour`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezcontour(fh)
```

When using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezcontour` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example, `k` in `myfun`:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then use an anonymous function to specify that parameter:

```
ezcontour(@(x,y)myfun(x,y,2))
```

Examples

The following mathematical expression defines a function of two variables, x and y.

$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}$$

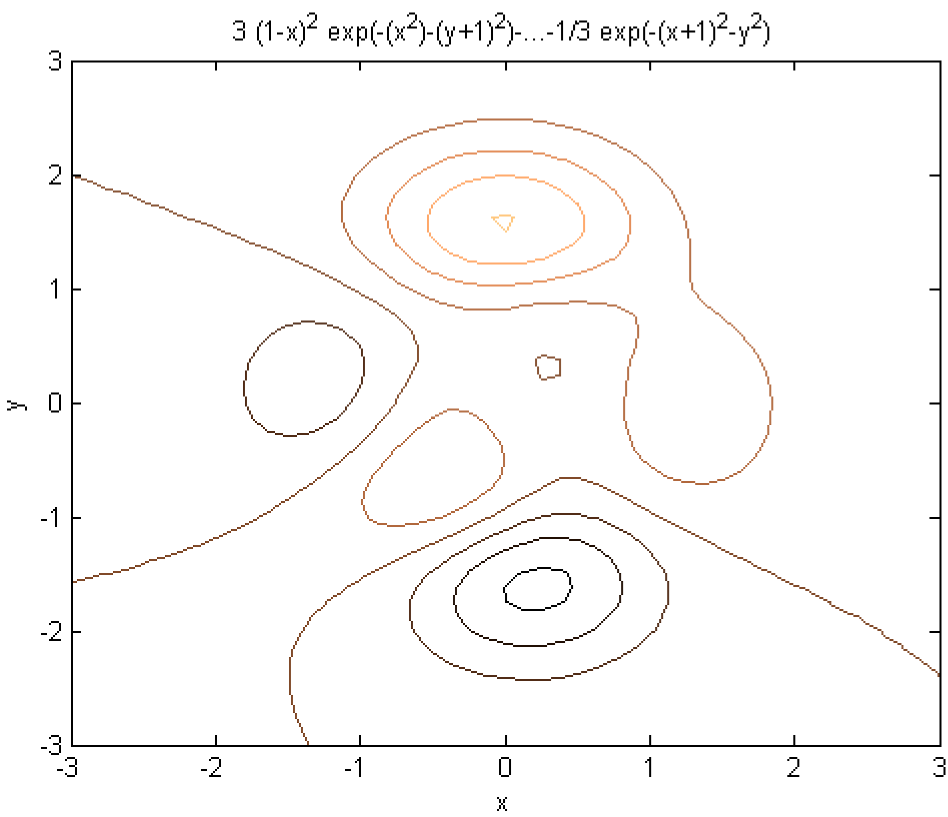
ezcontour requires a function handle argument that expresses this function using MATLAB syntax. This example uses an anonymous function, which you can define in the command window without creating a separate file.

```
f=@(x,y) 3*(1-x).^2.*exp(-(x.^2) - (y+1).^2) ...
   - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2) ...
   - 1/3*exp(-(x+1).^2 - y.^2);
```

For convenience, this function is written on three lines. The MATLAB peaks function evaluates this expression for different sizes of grids.

Pass the function handle f to ezcontour along with a domain ranging from -3 to 3 in both *x* and *y* and specify a computational grid of 49-by-49:

```
ezcontour(f,[-3,3],49)
```



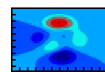
In this particular case, the title is too long to fit at the top of the graph, so MATLAB abbreviates the string.

See Also

`contour` | `ezcontourf` | `ezmesh` | `ezmeshc` | `ezplot` | `ezplot3` | `ezpolar` | `ezsurf` | `ezsurfc` | `function_handle`

Purpose

Easy-to-use filled contour plotter



Syntax

```
ezcontourf(fun)
ezcontourf(fun,domain)
ezcontourf(...,n)
ezcontourf(axes_handle,...)
h = ezcontourf(...)
```

Description

`ezcontourf(fun)` plots the contour lines of `fun(x,y)`using the `contourf` function. `fun` is plotted over the default domain: -2π < `x` < 2π, -2π < `y` < 2π.

`fun` can be a function handle or a string (see Tips).

`ezcontourf(fun,domain)` plots `fun(x,y)` over the specified `domain`. `domain` can be either a 4-by-1 vector `[xmin, xmax, ymin, ymax]` or a 2-by-1 vector `[min, max]`, where `min` < *x* < `max`, `min` < *y* < `max`).

`ezcontourf(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezcontourf(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = ezcontourf(...)` returns the handles to contour objects in `h`.

`ezcontourf` automatically adds a title and axis labels.

Tips

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the string expression you pass to `ezcontourf`. For example, the MATLAB syntax for a filled contour plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezcontourf('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezcontourf`.

If the function to be plotted is a function of the variables u and v (rather than x and y), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezcontourf('u^2 - v^3',[0,1],[3,6])` plots the contour lines for $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezcontourf`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezcontourf(fh)
```

When using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezcontourf` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example, `k` in `myfun`:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then you can use an anonymous function to specify that parameter:

```
ezcontourf(@(x,y)myfun(x,y,2))
```

Examples

The following mathematical expression defines a function of two variables, x and y.

$$f(x, y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - 10\left(\frac{x}{5} - x^3 - y^5\right)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}$$

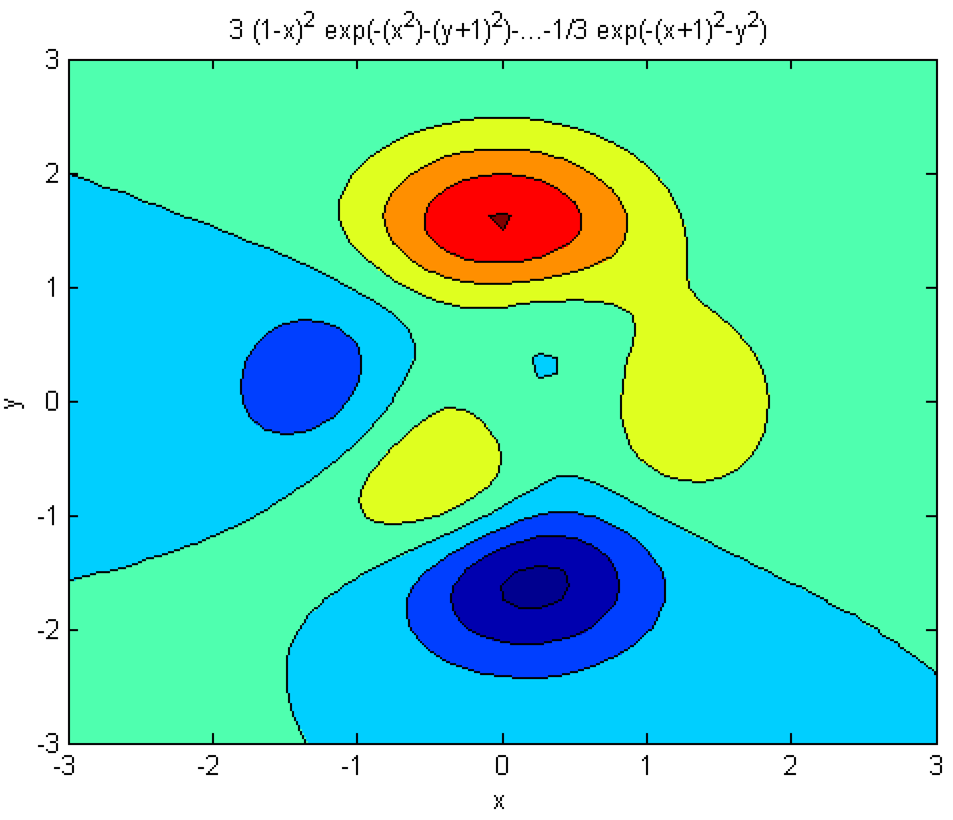
`ezcontourf` requires a string argument that expresses this function using MATLAB syntax to represent exponents, natural logs, etc. This function is represented by the string

```
f = @(x,y) 3*(1-x).^2.*exp(-(x.^2) - (y+1).^2) ...
        - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2) ...
        - 1/3*exp(-(x+1).^2 - y.^2);
```

For convenience, this string is written on three lines and concatenated into one string using square brackets.

Pass the string variable `f` to `ezcontourf` along with a domain ranging from -3 to 3 and specify a grid of 49-by-49:

```
ezcontourf(f,[-3,3],49)
```



In this particular case, the title is too long to fit at the top of the graph, so MATLAB abbreviates the string.

See Also

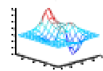
`contourf` | `ezcontour` | `ezmesh` | `ezmeshc` | `ezplot` | `ezplot3` | `ezpolar` | `ezsurf` | `ezsurfc` | `function_handle`

How To

- Anonymous Functions

Purpose

Easy-to-use 3-D mesh plotter



Syntax

```
ezmesh(fun)
ezmesh(fun,domain)
ezmesh(funx,funy,funz)
ezmesh(funx,funy,funz,[smin,smax,tmin,tmax])
ezmesh(funx,funy,funz,[min,max])
ezmesh(...,n)
ezmesh(...,'circ')
ezmesh(axes_handle,...)
h = ezmesh(...)
```

Description

`ezmesh(fun)` creates a graph of `fun(x,y)` using the `mesh` function. `fun` is plotted over the default domain: $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

`fun` can be a function handle or a string (see the Tips section).

`ezmesh(fun,domain)` plots `fun` over the specified `domain`. `domain` can be either a 4-by-1 vector [`xmin`, `xmax`, `ymin`, `ymax`] or a 2-by-1 vector [`min`, `max`] (where `min` < `x` < `max`, `min` < `y` < `max`).

`ezmesh(funx,funy,funz)` plots the parametric surface `funx(s,t)`, `funy(s,t)`, and `funz(s,t)` over the square: -2π < `s` < 2π, -2π < `t` < 2π.

`ezmesh(funx,funy,funz,[smin,smax,tmin,tmax])` or `ezmesh(funx,funy,funz,[min,max])` plots the parametric surface using the specified domain.

`ezmesh(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezmesh(...,'circ')` plots `fun` over a disk centered on the domain.

`ezmesh(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezmesh(...)` returns the handle to a surface object in `h`.

Tips

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the string expression you pass to `ezmesh`. For example, the MATLAB syntax for a mesh plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezmesh('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezmesh`.

If the function to be plotted is a function of the variables u and v (rather than x and y), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezmesh('u^2 - v^3',[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezmesh`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezmesh(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezmesh` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then you can use an anonymous function to specify that parameter:

```
ezmesh(@(x,y)myfun(x,y,2))
```

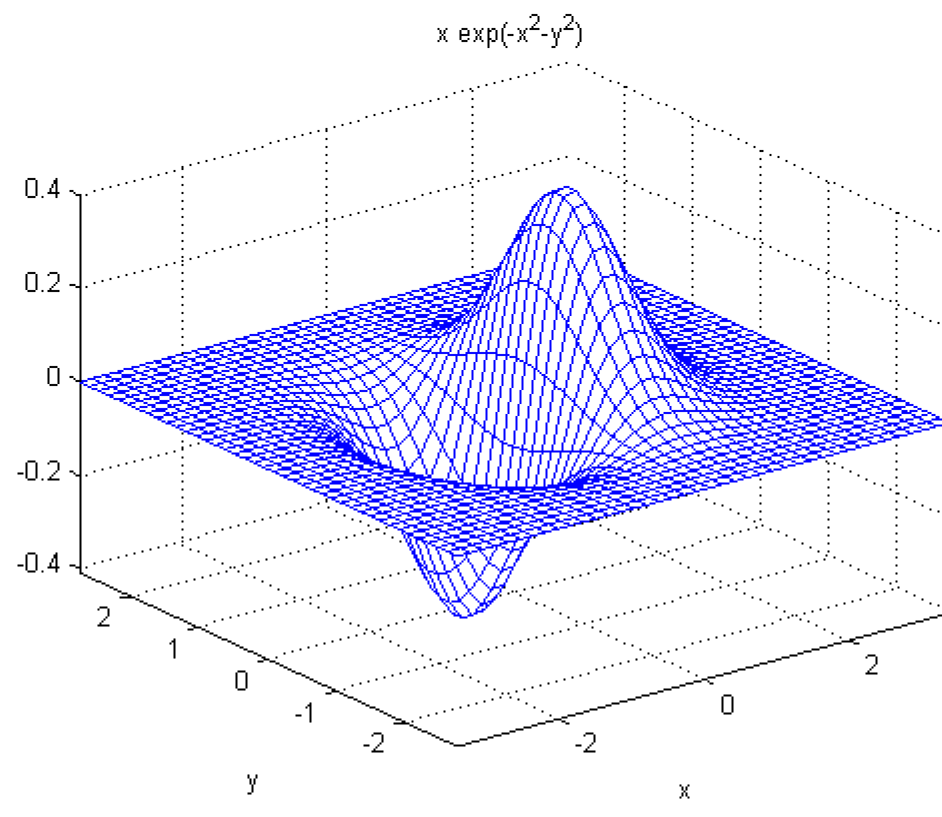
Examples

This example visualizes the function

$$f(x,y) = ex^{-x^2-y^2}$$

with a mesh plot drawn on a 40-by-40 grid. The mesh lines are set to a uniform blue color by setting the colormap to a single color:

```
fh = @(x,y) x.*exp(-x.^2-y.^2);
ezmesh(fh,40)
colormap([0 0 1])
```



See Also

`ezmeshc` | `function_handle` | `mesh`

How To

- Anonymous Functions

Purpose

Easy-to-use combination mesh/contour plotter

Syntax

```
ezmeshc(fun)
ezmeshc(fun,domain)
ezmeshc(funx,funy,funz)
ezmeshc(funx,funy,funz,[smin,smax,tmin,tmax])
ezmeshc(funx,funy,funz,[min,max])
ezmeshc(...,n)
ezmeshc(...,'circ')
ezmesh(axes_handle,...)
h = ezmeshc(...)
```

Description

`ezmeshc(fun)` creates a graph of `fun(x,y)` using the `meshc` function. `fun` is plotted over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

`fun` can be a function handle or a string (see the Tips section).

`ezmeshc(fun,domain)` plots `fun` over the specified `domain`. `domain` can be either a 4-by-1 vector [`xmin`, `xmax`, `ymin`, `ymax`] or a 2-by-1 vector [`min`, `max`] (where `min` < x < `max`, `min` < y < `max`).

`ezmeshc(funx,funy,funz)` plots the parametric surface `funx(s,t)`, `funy(s,t)`, and `funz(s,t)` over the square: -2π < `s` < 2π, -2π < `t` < 2π.

`ezmeshc(funx,funy,funz,[smin,smax,tmin,tmax])` or `ezmeshc(funx,funy,funz,[min,max])` plots the parametric surface using the specified domain.

`ezmeshc(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezmeshc(...,'circ')` plots `fun` over a disk centered on the domain.

`ezmesh(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezmeshc(...)` returns the handle to a surface object in `h`.

Tips

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the string expression you pass to `ezmeshc`. For example, the MATLAB syntax for a mesh/contour plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezmeshc('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezmeshc`.

If the function to be plotted is a function of the variables u and v (rather than x and y), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezmeshc('u^2 - v^3',[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezmeshc`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezmeshc(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezmeshc` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k)
z = x.^k - y.^k - 1;
```

then you can use an anonymous function to specify that parameter:

```
ezmeshc(@(x,y)myfun(x,y,2))
```

Examples

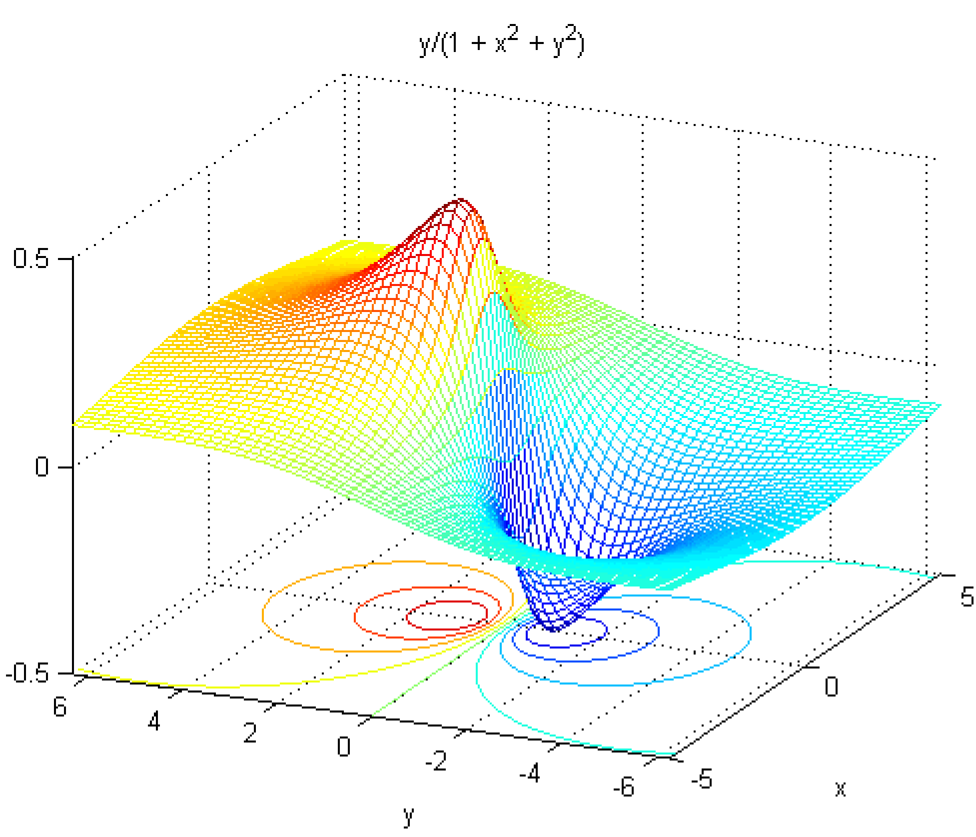
Create a mesh/contour graph of the expression

$$f(x,y) = \frac{y}{1 + x^2 + y^2}$$

over the domain $-5 < x < 5$, $-2*pi < y < 2*pi$:

```
ezmeshc('y/(1 + x^2 + y^2)',[-5,5,-2*pi,2*pi])
view(-65.5,26)
```

Use the mouse to rotate the axes to better observe the contour lines (this picture uses a view of azimuth = -65.5 and elevation = 26)



See Also

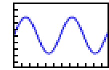
`ezmesh` | `ezsurfc` | `function_handle` | `meshc`

How To

- Anonymous Functions

Purpose

Easy-to-use function plotter



Syntax

```
ezplot(fun)
ezplot(fun,[xmin,xmax])
ezplot(fun2)
ezplot(fun2,[xymin,xymax])
ezplot(fun2,[xmin,xmax,ymin,ymax])
ezplot(funx,funy)
ezplot(funx,funy,[tmin,tmax])
ezplot(...,figure_handle)
ezplot(axes_handle,...)
h = ezplot(...)
```

Description

`ezplot(fun)` plots the expression `fun(x)` over the default domain $-2\pi < x < 2\pi$, where `fun(x)` is an explicit function of only x.

`fun` can be a function handle or a string.

`ezplot(fun,[xmin,xmax])` plots `fun(x)` over the domain: `xmin` < x < `xmax`.

For an implicit function, `fun2(x,y)`:

`ezplot(fun2)` plots `fun2(x,y) = 0` over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

`ezplot(fun2,[xymin,xymax])` plots `fun2(x,y) = 0` over `xymin` < `x` < `xymax` and `xymin` < `y` < `xymax`.

`ezplot(fun2,[xmin,xmax,ymin,ymax])` plots `fun2(x,y) = 0` over `xmin` < `x` < `xmax` and `ymin` < `y` < `ymax`.

`ezplot(funx,funy)` plots the parametrically defined planar curve `funx(t)` and `funy(t)` over the default domain 0 < `t` < 2π.

`ezplot(funx,funy,[tmin,tmax])` plots `funx(t)` and `funy(t)` over `tmin` < `t` < `tmax`.

`ezplot(...,figure_handle)` plots the given function over the specified domain in the figure window identified by the handle `figure`.

`ezplot(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezplot(...)` returns the handle to all the plot objects in h.

Tips

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezplot`. For example, the MATLAB syntax for a plot of the expression

```
x.^2 - y.^2
```

which represents an implicitly defined function, is written as

```
ezplot('x^2 - y^2')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezplot`.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezplot`,

```
fh = @(x,y) sqrt(x.^2 + y.^2 - 1);
ezplot(fh)
axis equal
```

which plots a circle. Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezplot` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k)
```

```
z = x.^k - y.^k - 1;
```

then you can use an anonymous function to specify that parameter:

```
ezplot(@(x,y)myfun(x,y,2))
```

Controlling Line Color

`ezplot` colors lines according to the type of graphics object it chooses to plot the input function.

- For an implicit function, ezplot uses a `LineSeries` object to generate the graph. To change the color, you should set the `Color` property.
- For an explicit function, the graph generated is a `Hggroup` object. So, to change the color, you should set the `LineColor` property.

Example

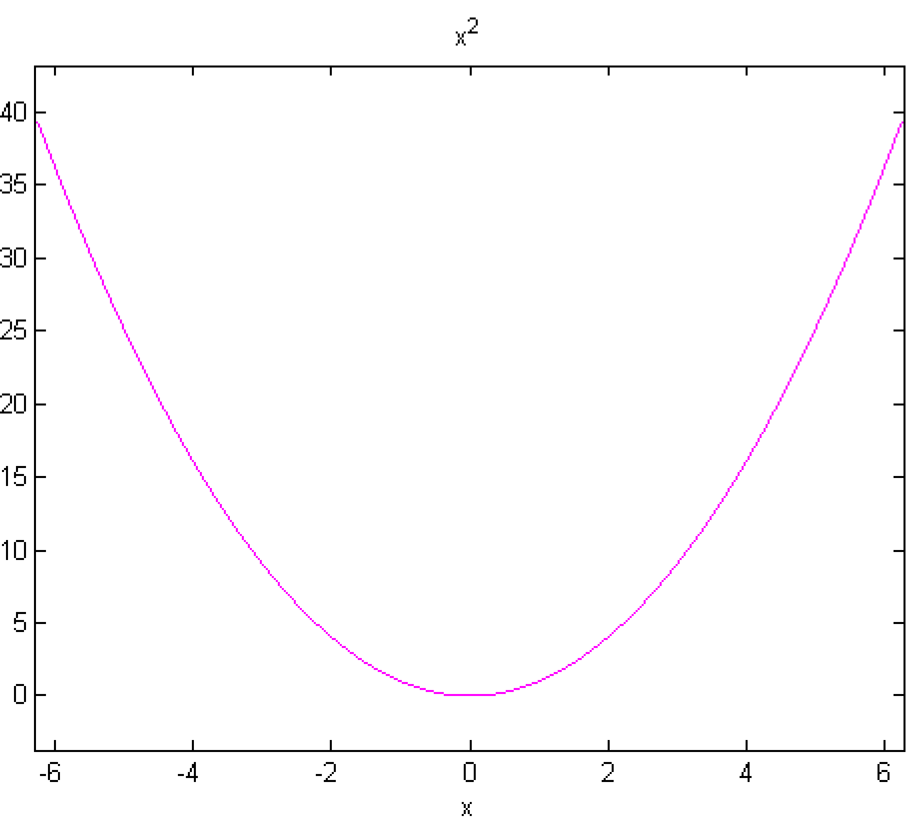
Plotting an Explicit Function

This example plots an explicit function:

x^2

over the domain [-2π, 2π].

```
h = ezplot('x^2')
set(h, 'Color', 'm');   % Make the line magenta
```



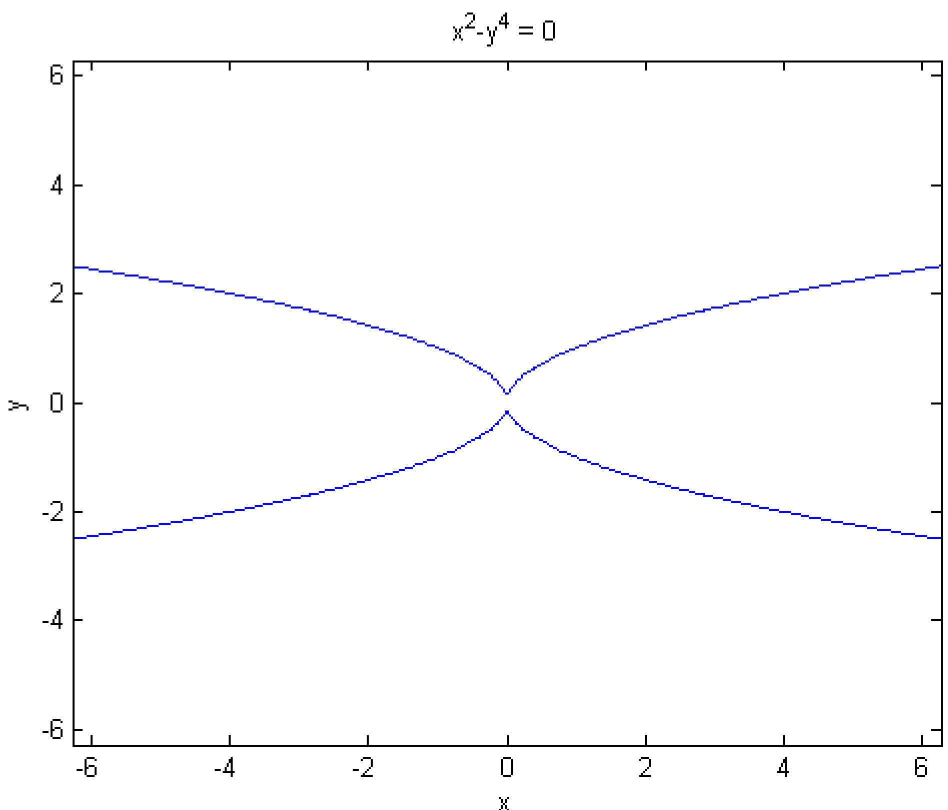
Plotting an Implicit Function

This example plots the implicitly defined function

$x^2 - y^4 = 0$

over the domain [-2π, 2π].

```
ezplot('x^2-y^4')
colormap([0 0 1])   % Make the line blue
```



See Also

`ezplot3` | `ezpolar` | `function_handle` | `plot`

How To

- Anonymous Functions

ezplot3

Purpose

Easy-to-use 3-D parametric curve plotter

Syntax

```
ezplot3(funx,funy,funz)
ezplot3(funx,funy,funz,[tmin,tmax])
ezplot3(...,'animate')
ezplot3(axes_handle,...)
h = ezplot3(...)
```

Description

`ezplot3(funx,funy,funz)` plots the spatial curve `funx(t)`, `funy(t)`, and `funz(t)` over the default domain 0 < `t` < 2π.

`funx`, `funy`, and `funz` can be function handles or strings (see the Tips section).

`ezplot3(funx,funy,funz,[tmin,tmax])` plots the curve `funx(t)`, `funy(t)`, and `funz(t)` over the domain `tmin` < `t` < `tmax`.

`ezplot3(...,'animate')` produces an animated trace of the spatial curve.

`ezplot3(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezplot3(...)` returns the handle to the plotted objects in `h`.

Tips

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezplot3`. For example, the MATLAB syntax for a plot of the expression

```
x = s./2, y = 2.*s, z = s.^2;
```

which represents a parametric function, is written as

```
ezplot3('s/2','2*s','s^2')
```

That is, `s/2` is interpreted as `s./2` in the string you pass to `ezplot3`.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezplot3`.

```
fh1 = @(s) s./2; fh2 = @(s) 2.*s; fh3 = @(s) s.^2;
ezplot3(fh1,fh2,fh3)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezplot` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfuntk`:

```
function s = myfuntk(t,k)
s = t.^k.*sin(t);
```

then you can use an anonymous function to specify that parameter:

```
ezplot3(@cos,@(t)myfuntk(t,1),@sqrt)
```

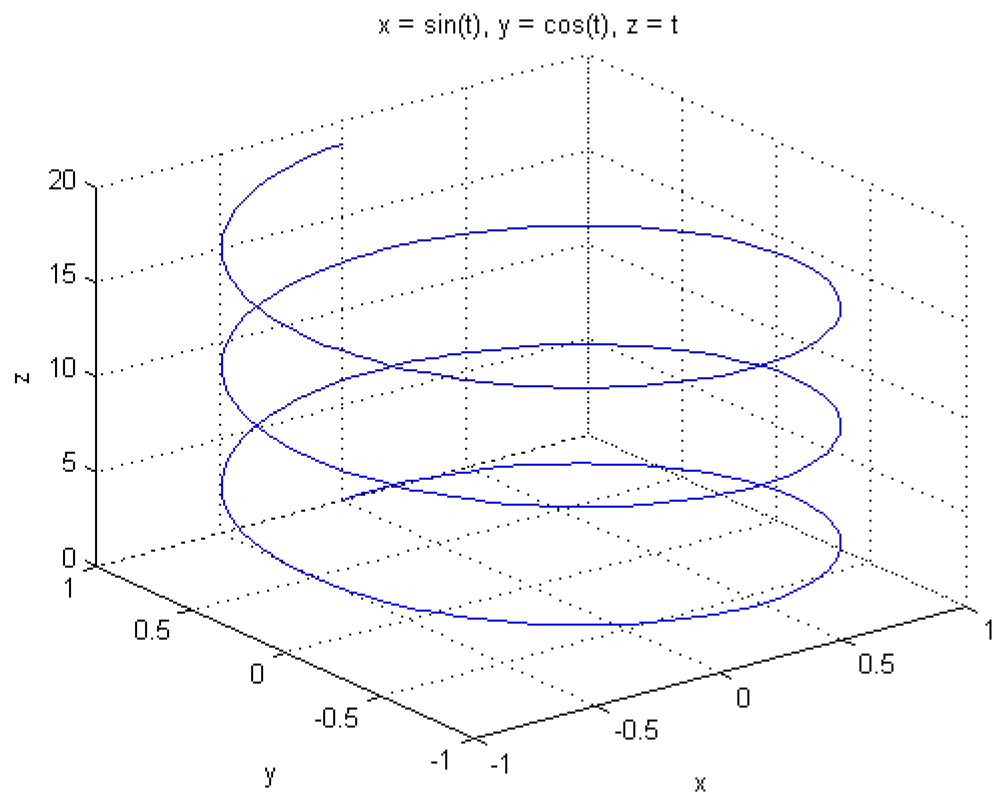
Examples

This example plots the parametric curve

$$x = \sin t, \quad y = \cos t, \quad z = t$$

over the domain [0,6π]:

```
ezplot3('sin(t)','cos(t)','t',[0,6*pi])
```



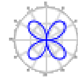
See Also `ezplot` | `ezpolar` | `function_handle` | `plot3`

How To

- Anonymous Functions

Purpose

Easy-to-use polar coordinate plotter



Syntax

```
ezpolar(fun)
ezpolar(fun,[a,b])
ezpolar(axes_handle,...)
h = ezpolar(...)
```

Description

`ezpolar(fun)` plots the polar curve `rho = fun(theta)` over the default domain 0 < `theta` < 2π.

`fun` can be a function handle or a string (see the Tips section).

`ezpolar(fun,[a,b])` plots `fun` for `a` < `theta` < `b`.

`ezpolar(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezpolar(...)` returns the handle to a line object in `h`.

Tips

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezpolar`. For example, the MATLAB syntax for a plot of the expression

```
t.^2.*cos(t)
```

which represents an implicitly defined function, is written as

```
ezpolar('t^2*cos(t)')
```

That is, `t^2` is interpreted as `t.^2` in the string you pass to `ezpolar`.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezpolar`.

```
fh = @(t) t.^2.*cos(t);
ezpolar(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezpolar` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k1` and `k2` in `myfun`:

```
function s = myfun(t,k1,k2)
s = sin(k1*t).*cos(k2*t);
```

then you can use an anonymous function to specify the parameters:

```
ezpolar(@(t)myfun(t,2,3))
```

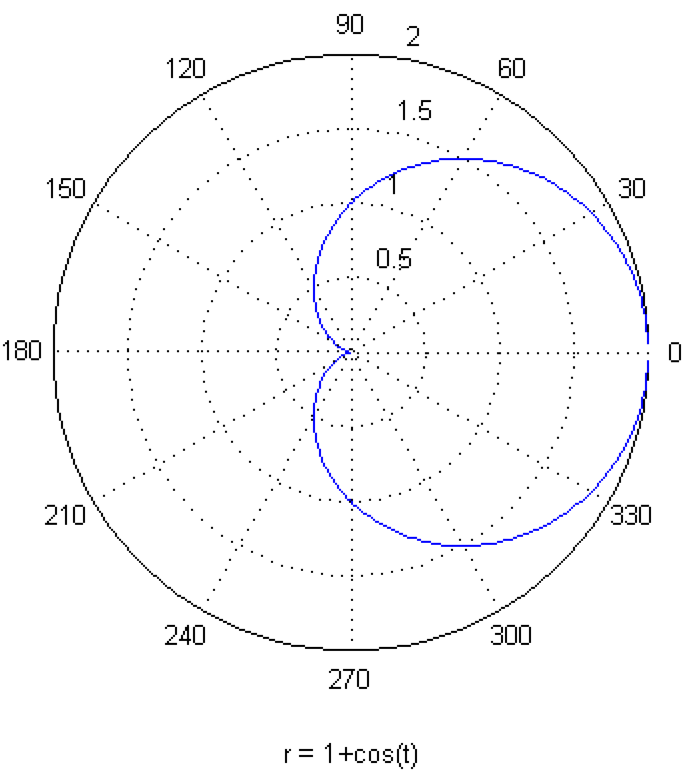
Examples

This example creates a polar plot of the function

$1 + cos(t)$

over the domain $[0, 2\pi]$:

```
ezpolar('1+cos(t)')
```



See Also

`ezplot` | `ezplot3` | `function_handle` | `plot` | `plot3` | `polar`

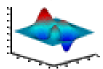
How To

- Anonymous Functions

ezsurf

Purpose

Easy-to-use 3-D colored surface plotter



Syntax

```
ezsurf(fun)
ezsurf(fun,domain)
ezsurf(funx,funy,funz)
ezsurf(funx,funy,funz,[smin,smax,tmin,tmax])
ezsurf(funx,funy,funz,[min,max])
ezsurf(...,n)
ezsurf(...,'circ')
ezsurf(axes_handle,...)
h = ezsurf(...)
```

Description

`ezsurf(fun)` creates a graph of `fun(x,y)` using the `surf` function. `fun` is plotted over the default domain: $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$.

`fun` can be a function handle or a string (see the Tips section).

`ezsurf(fun,domain)` plots `fun` over the specified `domain`. `domain` must be a vector. See the “Algorithms” on page 1-1582 section for details on vector inputs vs axes limit outputs.

`ezsurf(funx,funy,funz)` plots the parametric surface `funx(s,t)`, `funy(s,t)`, and `funz(s,t)` over the square: $-2\pi < s < 2\pi$, $-2\pi < t < 2\pi$.

`ezsurf(funx,funy,funz,[smin,smax,tmin,tmax])` or `ezsurf(funx,funy,funz,[min,max])` plots the parametric surface using the specified domain.

`ezsurf(...,n)` plots `fun` over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezsurf(...,'circ')` plots `fun` over a disk centered on the domain.

`ezsurf(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezsurf(...)` returns the handle to a surface object in `h`.

Tips

`ezsurf` and `ezsurfc` do not accept complex inputs.

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezmesh`. For example, the MATLAB syntax for a surface plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezsurf('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezsurf`.

If the function to be plotted is a function of the variables u and v (rather than x and y), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezsurf('u^2 - v^3',[0,1],[3,6])` plots u^2 - v^3 over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezsurf`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezsurf(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezsurf` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k1,k2,k3)
z = x.*(y.^k1)./(x.^k2 + y.^k3);
```

then you can use an anonymous function to specify that parameter:

```
ezsurf(@(x,y)myfun(x,y,2,2,4))
```

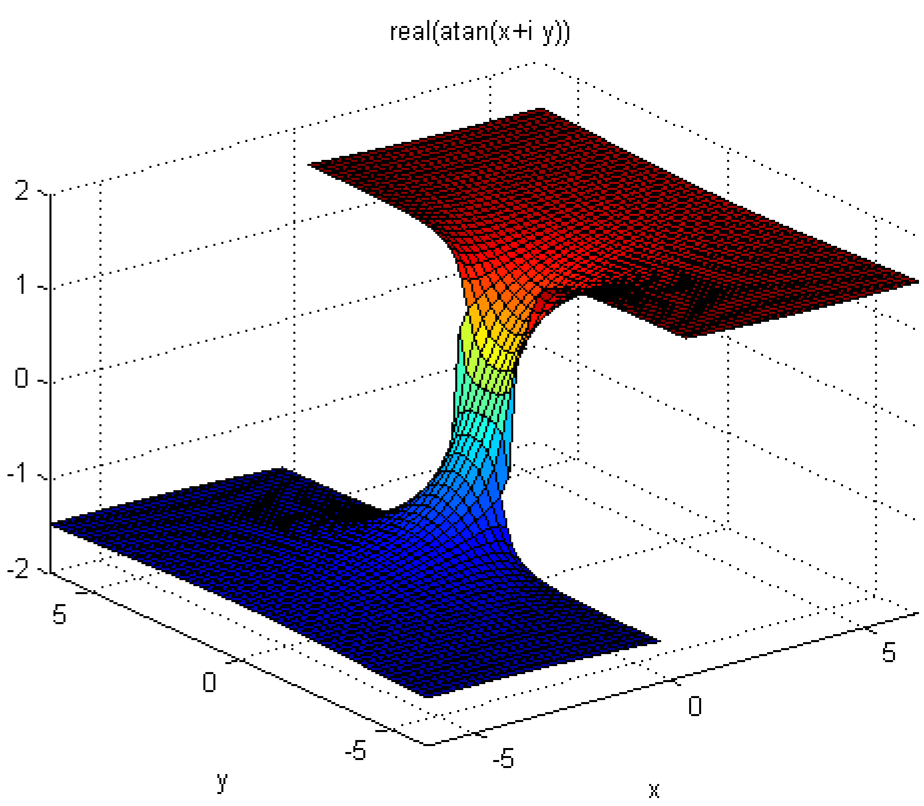
Examples

ezsurf does not graph points where the mathematical function is not defined (these data points are set to NaNs, which do not plot). This example illustrates this filtering of singularities/discontinuous points by graphing the function

$$f(x,y) = real\left(a\tan\left(x+iy\right)\right)$$

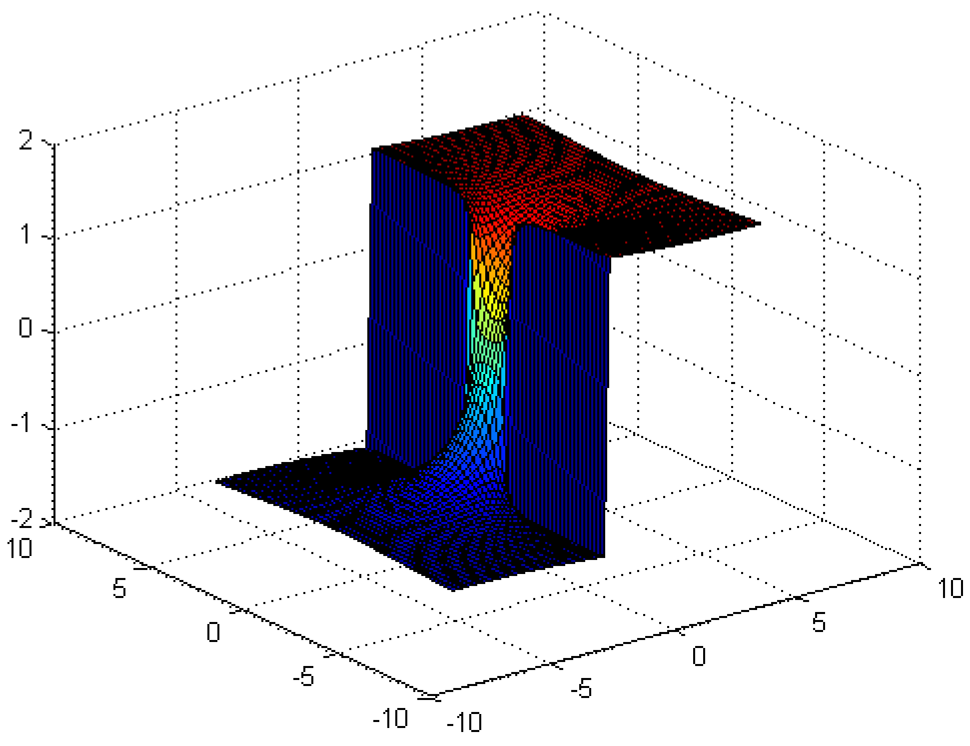
over the default domain $-2\pi < x < 2\pi$, $-2\pi < y < 2\pi$:

```
ezsurf('real(atan(x+i*y))')
```



Using `surf` to plot the same data produces a graph without filtering of discontinuities (as well as requiring more steps):

```
[x,y] = meshgrid(linspace(-2*pi,2*pi,60));
z = real(atan(x+1i.*y));
surf(x,y,z)
```



Note also that `ezsurf` creates graphs that have axis labels, a title, and extend to the axis limits.

Algorithms

`ezsurf` determines the *x*- and *y-axes* limits in different ways depending on how you input the domain (if at all). In the following table, `R` is the vector [`xmin`, `xmax`, `ymin`, `ymax`] and `v` is the manually entered domain vector.

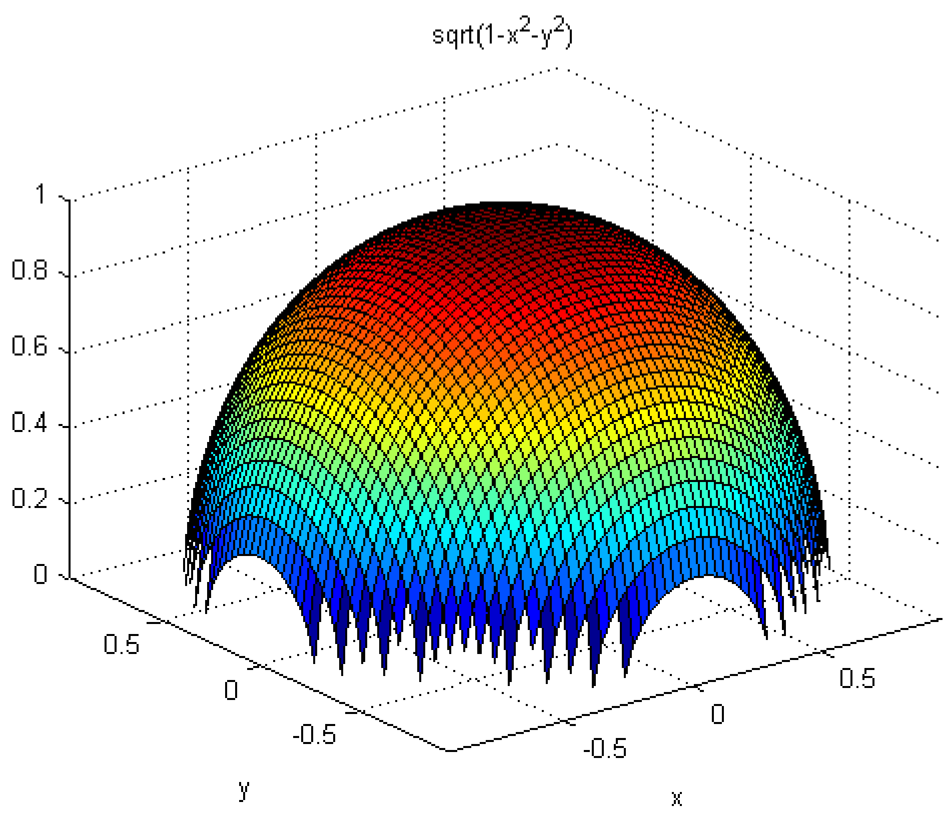
Number of domain values specified:	Resulting domain vector:
`v = [ ];`	`R = [-2*pi, 2*pi, -2*pi, 2*pi];`
`v = [ v(1) ];`	`R = double([-abs(v),abs(v),-abs(v),abs(v)]);`
`v = [ v(1) v(2) ];`	`R = double([v(1),v(2),v(1),v(2)]);`
`v = [ v(1) v(2) v(3) ];`	`R = double([-v(1),v(2),-abs(v(3)),abs(v(3))]);`
`v = [ v(1) v(2) v(3) v(4) ];`	`R = double(v);`
`v = [ v(1)..v(n) ]; n>4`	`R = double([-abs(v(1)), abs(v(1)), -abs(v(1)), abs(`

If you specify a single number in non-vector format (without square brackets, [ ]), `ezsurf` interprets it as the `n`, the number of points desired between the axes `max` and `min` values.

By default, `ezsurf` uses 60 points between the `max` and `min` values of an axes. When the `min` and `max` values are the default values (`R = [-2*pi, 2*pi, -2*pi, 2*pi];`), `ezsurf` ensures the 60 points fall within the non-complex range of the specified equation. For example,

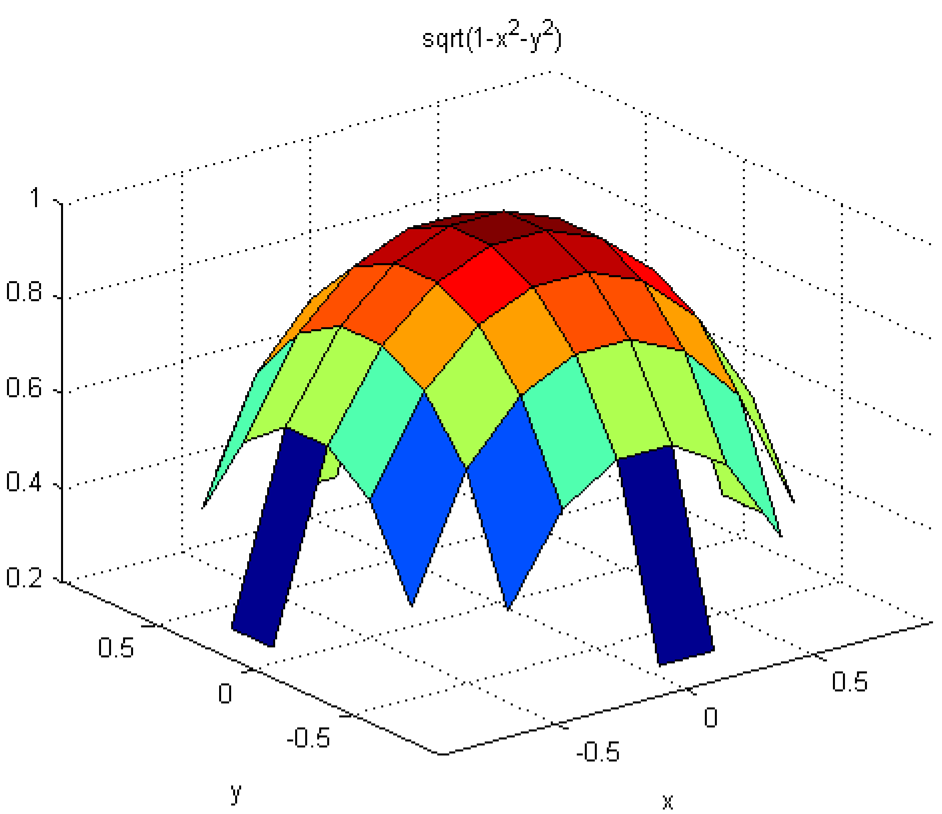
$\sqrt{1-x^2-y^2}$ is only real when $x^2-y^2 \le 1$. The default graph of this function looks like this:

```
ezsurf('sqrt(1 x^2 y^2)')
```



You can see that there are 60 points between the minimum and maximum values for which $\sqrt{1-x^2-y^2}$ has real values. However, when you specify the domain values to be the same as the default (`R = [-2*pi, 2*pi, -2*pi, 2*pi];`), a different result appears:

```
ezsurf('sqrt(1 x^2 y^2)',[-2*pi 2*pi])
```



In this case, the graphic limits are the same, but `ezsurf` used 60 points between the user-defined limits instead of checking to see if all those points would have real answers.

See Also

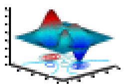
`ezmesh` | `ezsurfc` | `function_handle` | `surf`

How To

- Anonymous Functions

Purpose

Easy-to-use combination surface/contour plotter



Syntax

```
ezsurfc(fun)
ezsurfc(fun,domain)
ezsurfc(funx,funy,funz)
ezsurfc(funx,funy,funz,[smin,smax,tmin,tmax])
ezsurfc(funx,funy,funz,[min,max])
ezsurfc(...,n)
ezsurfc(...,'circ')
ezsurfc(axes_handle,...)
h = ezsurfc(...)
```

Description

`ezsurfc(fun)` creates a graph of `fun(x,y)` using the `surfc` function. The function `fun` is plotted over the default domain: -2π < `x` < 2π, -2π < `y` < 2π.

`fun` can be a function handle or a string (see the Tips section).

`ezsurfc(fun,domain)` plots `fun` over the specified `domain`. `domain` can be either a 4-by-1 vector [`xmin`, `xmax`, `ymin`, `ymax`] or a 2-by-1 vector [`min`, `max`] (where `min` < `x` < `max`, `min` < `y` < `max`).

`ezsurfc(funx,funy,funz)` plots the parametric surface `funx(s,t)`, `funy(s,t)`, and `funz(s,t)` over the square: -2π < `s` < 2π, -2π < `t` < 2π.

`ezsurfc(funx,funy,funz,[smin,smax,tmin,tmax])` or `ezsurfc(funx,funy,funz,[min,max])` plots the parametric surface using the specified domain.

`ezsurfc(...,n)` plots *f* over the default domain using an `n`-by-`n` grid. The default value for `n` is 60.

`ezsurfc(...,'circ')` plots *f* over a disk centered on the domain.

`ezsurfc(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = ezsurfc(...)` returns the handles to the graphics objects in `h`.

Tips

`ezsurf` and `ezsurfc` do not accept complex inputs.

Passing the Function as a String

Array multiplication, division, and exponentiation are always implied in the expression you pass to `ezsurfc`. For example, the MATLAB syntax for a surface/contour plot of the expression

```
sqrt(x.^2 + y.^2);
```

is written as

```
ezsurfc('sqrt(x^2 + y^2)')
```

That is, `x^2` is interpreted as `x.^2` in the string you pass to `ezsurfc`.

If the function to be plotted is a function of the variables u and v (rather than x and y), then the domain endpoints `umin`, `umax`, `vmin`, and `vmax` are sorted alphabetically. Thus, `ezsurfc('u^2 - v^3',[0,1],[3,6])` plots $u^2 - v^3$ over $0 < u < 1$, $3 < v < 6$.

Passing a Function Handle

Function handle arguments must point to functions that use MATLAB syntax. For example, the following statements define an anonymous function and pass the function handle `fh` to `ezsurfc`.

```
fh = @(x,y) sqrt(x.^2 + y.^2);
ezsurf(fh)
```

Note that when using function handles, you must use the array power, array multiplication, and array division operators (`.^, .*, ./`) since `ezsurfc` does not alter the syntax, as in the case with string inputs.

Passing Additional Arguments

If your function has additional parameters, for example `k` in `myfun`:

```
function z = myfun(x,y,k1,k2,k3)
z = x.*(y.^k1)./(x.^k2 + y.^k3);
```

then you can use an anonymous function to specify that parameter:

```
ezsurfc(@(x,y)myfun(x,y,2,2,4))
```

Examples

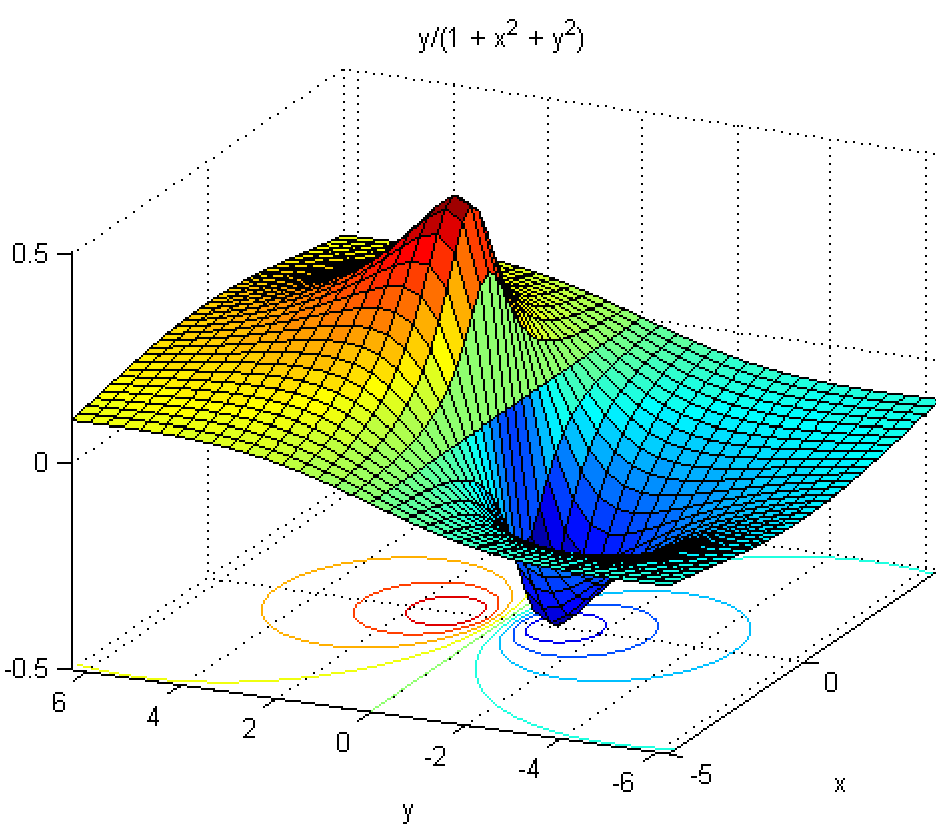
Create a surface/contour plot of the expression

$$f(x,y) = \frac{y}{1 + x^2 + y^2}$$

over the domain -5 < x < 5, -2*pi < y < 2*pi, with a computational grid of size 35-by-35:

```
ezsurfc('y/(1 + x^2 + y^2)',[-5,5,-2*pi,2*pi],35)
```

Put the plot in `rotate3d` mode to use the mouse to rotate the axes to better observe the contour lines (this picture uses a view of azimuth = -65.5 and elevation = 26).



See Also

ezmesh | ezmeshc | ezsurf | function_handle | surfc

How To

- Anonymous Functions

TriRep.faceNormals

Purpose

(Will be removed) Unit normals to specified triangles

Note `faceNormals(TriRep)` will be removed in a future release. Use `faceNormal(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
FN = faceNormals(TR, TI)
```

Description

`FN = faceNormals(TR, TI)` returns the unit normal vector to each of the specified triangles TI.

Note This query is only applicable to triangular surface meshes.

Input Arguments

`TR`	Triangulation representation.
`TI`	Column vector of indices that index into the triangulation matrix `TR.Triangulation`.

Output Arguments

`FN` m-by-3 matrix. `m = length(TI)`, the number of triangles to be queried. Each row `FN(i,:)` represents the unit normal vector to triangle `TI(i)`.

If TI is not specified the unit normal information for the entire triangulation is returned, where the normal associated with triangle i is the `i`'th row of `FN`.

Examples

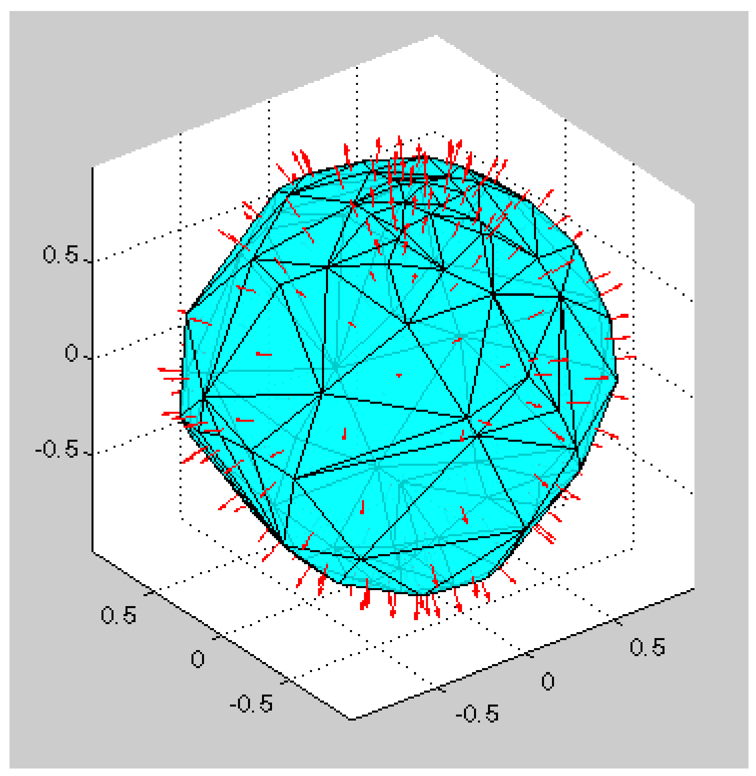
Triangulate a sample of random points on the surface of a sphere and use the `TriRep` to compute the normal to each triangle:

```
numpts = 100;
```

```
thetha = rand(numpts,1)*2*pi;
phi = rand(numpts,1)*pi;
x = cos(thetha).*sin(phi);
y = sin(thetha).*sin(phi);
z = cos(phi);
dt = DelaunayTri(x,y,z);
[tri Xb] = freeBoundary(dt);
tr = TriRep(tri, Xb);
P = incenters(tr);
fn = faceNormals(tr);
trisurf(tri,Xb(:,1),Xb(:,2),Xb(:,3), ...
     'FaceColor', 'cyan', 'faceAlpha', 0.8);
axis equal;
hold on;
```

Display the result using a quiver plot:

```
quiver3(P(:,1),P(:,2),P(:,3), ...
     fn(:,1),fn(:,2),fn(:,3),0.5, 'color','r');
hold off;
```



See Also `freeBoundary` | `delaunayTriangulation` | `triangulation`

Purpose

Prime factors

Syntax

```
f = factor(n)
```

Description

`f = factor(n)` returns a row vector containing the prime factors of `n`. Vector `f` is of the same data type as `n`.

Input Arguments

n - Input values

scalar, real, nonnegative integer values

Input values, specified as scalars that are real, nonnegative, and integer-valued.

Example: `10`

Example: `int16(64)`

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Examples

Prime Factors of Double Integer Value

```
f = factor(200)
```

```
f =

     2     2     2     5     5
```

Multiply the elements of `f` to reproduce the input value.

```
prod(f)
```

```
ans =

   200
```

Prime Factors of Unsigned Integer Value

```
n = uint16(138);
```

```
f = factor(n)
```

```
f =

     2     3    23
```

Multiply the elements of `f` to reproduce `n`.

```
prod(f)
```

```
ans =

   138
```

See Also `isprime` | `primes`

Purpose

Factorial of input

Syntax

```
f = factorial(n)
```

Description

`f = factorial(n)` returns the product of all positive integers less than or equal to `n`, where `n` is a nonnegative integer value. If `n` is an array, then `f` contains the factorial of each value of `n`. The data type and size of `f` is the same as that of `n`.

Input Arguments

n - Input values
scalar, vector, or array of real, nonnegative integer values

Input values, specified as a scalar, vector, or array of real, nonnegative integers.

Example: `5`

Example: `[0 1 2 3 4]`

Example: `int16([10 15 20])`

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

Limitations

- The result is only accurate for double-precision values of `n` that are less than or equal to `21`. A larger value of `n` produces a result that has the correct order of magnitude and is accurate for the first 15 digits. This is because double-precision numbers are only accurate up to 15 digits.
- For single-precision input, the result is only accurate for values of `n` that are less than or equal to `13`. A larger value of `n` produces a result that has the correct order of magnitude and is accurate for the first 8 digits. This is because single-precision numbers are only accurate up to 8 digits.

Examples

10!

```
f = factorial(10)
```

```
f =

     3628800
```

22!

```
format long
f = factorial(22)
```

```
f =

     1.124000727777608e+21
```

In this case, `f` is accurate up to 15 digits, `1.12400072777760e+21`, because double-precision numbers are only accurate up to 15 digits.

Reset the output format to the default.

```
format
```

Factorial of Array Elements

```
n = [0 1 2; 3 4 5];
f = factorial(n)
```

```
f =

     1     1     2
     6    24   120
```

Factorial of Unsigned Integer Values

```
n = uint64([5 10 15 20]);
f = factorial(n)
```

```
f =

    120      3628800      1307674368000      2432902008176640000
```

See Also `prod`

false

Purpose

Logical 0 (false)

Syntax

```
false
false(n)
false(m, n)
false(m, n, p, ...)
false(size(A))
false(...,'like',p)
```

Description

`false` is shorthand for `logical(0)`.

`false(n)` is an `n`-by-`n` matrix of logical zeros.

`false(m, n)` or `false([m, n])` is an `m`-by-`n` matrix of logical zeros.

`false(m, n, p, ...)` or `false([m n p ...])` is an `m`-by-`n`-by-`p`-by-`...` array of logical zeros.

Note The size inputs `m`, `n`, `p`, `...` should be nonnegative integers. Negative integers are treated as 0.

`false(size(A))` is an array of logical zeros that is the same size as array `A`.

`false(...,'like',p)` is an array of logical zeros of the same data type and sparsity as the logical array `p`.

Tips

`false(n)` is much faster and more memory efficient than `logical(zeros(n))`.

See Also

`true` | `logical`

Purpose

Close one or all open files

Syntax

```
fclose(fileID)
fclose('all')
status = fclose(...)
```

Description

`fclose(fileID)` closes an open file. *`fileID`* is an integer file identifier obtained from `fopen`.

`fclose('all')` closes all open files.

`status` `= fclose(...)` returns a *`status`* of `0` when the close operation is successful. Otherwise, it returns `-1`.

See Also

`ferror` | `fopen` | `frewind` | `fseek` | `ftell` | `feof` | `fscanf` | `fprintf` | `fread` | `fwrite`

fclose (serial)

Purpose

Disconnect serial port object from device

Syntax

```
fclose(obj)
```

Description

`fclose(obj)` disconnects `obj` from the device, where `obj` is a serial port object or an array of serial port objects.

Tips

If `obj` was successfully disconnected, then the `Status` property is configured to `closed` and the `RecordStatus` property is configured to `off`. You can reconnect `obj` to the device using the `fopen` function.

An error is returned if you issue `fclose` while data is being written asynchronously. In this case, you should abort the write operation with the `stopasync` function, or wait for the write operation to complete.

If you use the `help` command to display help for `fclose`, then you need to supply the pathname shown below.

```
help serial/fclose
```

Examples

This example creates the serial port object `s` on a Windows platform, connects `s` to the device, writes and reads text data, and then disconnects `s` from the device using `fclose`.

```
s = serial('COM1');
fopen(s)
fprintf(s, '*IDN?')
idn = fscanf(s);
fclose(s)
```

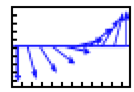
At this point, the device is available to be connected to a serial port object. If you no longer need `s`, you should remove from memory with the `delete` function, and remove it from the workspace with the `clear` command.

See Also

`clear` | `delete` | `fopen` | `stopasync` | `RecordStatus` | `Status`

Purpose

Plot velocity vectors



Syntax

```
feather(U,V)
feather(Z)
feather(...,LineSpec)
feather(axes_handle,...)
h = feather(...)
```

Description

A feather plot displays vectors emanating from equally spaced points along a horizontal axis. You express the vector components relative to the origin of the respective vector.

`feather(U,V)` displays the vectors specified by `U` and `V`, where `U` contains the *x* components as relative coordinates, and `V` contains the *y* components as relative coordinates.

`feather(Z)` displays the vectors specified by the complex numbers in `Z`. This is equivalent to `feather(real(Z),imag(Z))`.

`feather(...,LineSpec)` draws a feather plot using the line type, marker symbol, and color specified by `LineSpec`.

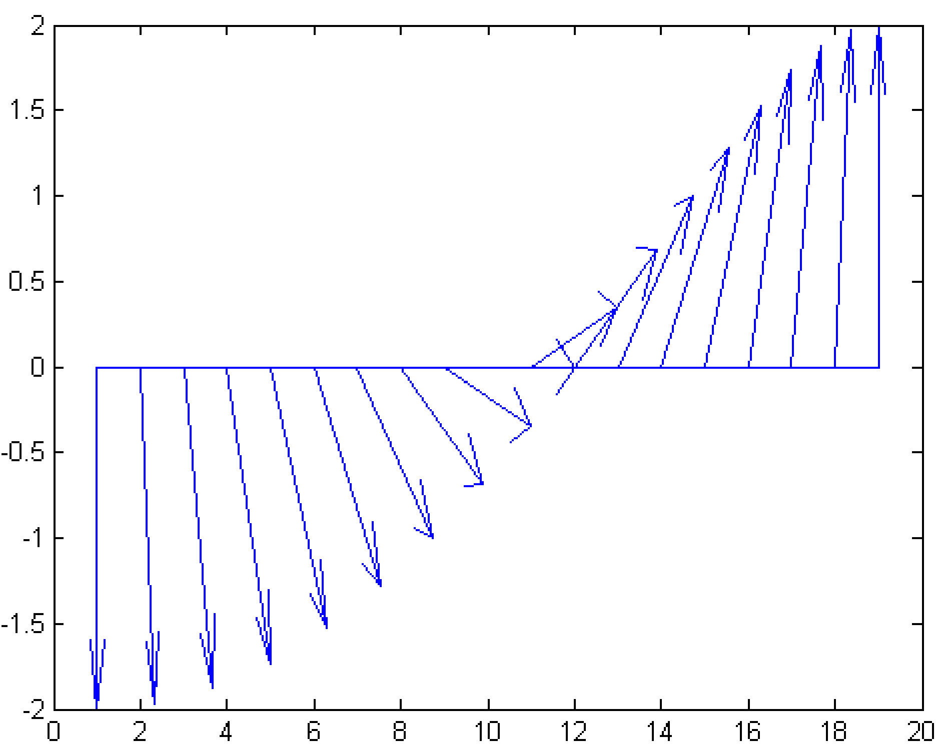
`feather(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = feather(...)` returns the handles to line objects in `h`.

Examples

Create a feather plot showing the direction of `theta`.

```
theta = (-90:10:90)*pi/180;
r = 2*ones(size(theta));
[u,v] = pol2cart(theta,r);
feather(u,v);
```



See Also

`compass` | `LineSpec` | `rose`

Purpose

(Will be removed) Sharp edges of surface triangulation

Note `featureEdges(TriRep)` will be removed in a future release. Use `featureEdges(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
FE = featureEdges(TR, filterangle)
```

Description

`FE = featureEdges(TR, filterangle)` returns an edge matrix `FE`. This method is typically used to extract the sharp edges in the surface mesh for the purpose of display. Edges that are shared by only one triangle and edges that are shared by more than two triangles are considered to be feature edges by default.

Note This query is only applicable to triangular surface meshes.

Input Arguments

`TR`	Triangulation representation.
`filterangle`	The threshold angle in radians. Must be in the range $(0, \pi)$. `featureEdges` will return adjacent triangles that have a dihedral angle that deviates from π by an angle greater than `filterangle`.

Output Arguments

`FE`	Edges of the triangulation. `FE` is of size `m`-by-2 where `m` is the number of computed feature edges in the mesh. The vertices of the edges index into the array of points representing the vertex coordinates, `TR.X`.

Examples

Create a surface triangulation:

```
x = [0 0 0 0 0 3 3 3 3 3 3 6 6 6 6 6 9 9 9 9 9 9]';
y = [0 2 4 6 8 0 1 3 5 7 8 0 2 4 6 8 0 1 3 5 7 8]';
dt = DelaunayTri(x,y);
tri = dt(:,:);
```

Elevate the 2-D mesh to create a surface:

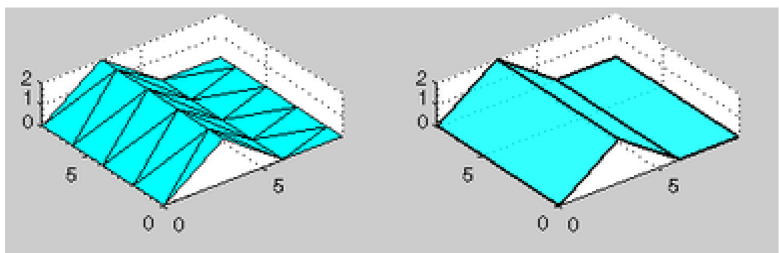
```
z = [0 0 0 0 0 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0]';
subplot(1,2,1);
trisurf(tri,x,y,z, 'FaceColor', 'cyan');
axis equal;
% TRISURF display of surface mesh
% showing mesh edges
```

Compute the feature edges using a filter angle of `pi/6`:

```
tr = TriRep(tri, x,y,z);
fe = featureEdges(tr,pi/6)';
subplot(1,2,2);
trisurf(tr, 'FaceColor', 'cyan', 'EdgeColor','none', ...
     'FaceAlpha', 0.8); axis equal;
```

Add the feature edges to the plot:

```
hold on;
plot3(x(fe), y(fe), z(fe), 'k', 'LineWidth',1.5);
hold off;
% TRISURF display of surface mesh
% suppressing mesh edges
% and showing feature edges
```



See Also

`edges` | `delaunayTriangulation` | `triangulation`

feof

Purpose

Test for end-of-file

Syntax

`status = feof(fileID)`

Description

`status` `= feof(`*`fileID`*`)` returns `1` if a previous operation set the end-of-file indicator for the specified file. Otherwise, `feof` returns `0`. *`fileID`* is an integer file identifier obtained from `fopen`.

Opening an empty file does *not* set the end-of-file indicator. Read operations, and the `fseek` and `frewind` functions, move the file position indicator.

Examples

Read `bench.dat`, which contains MATLAB benchmark data, one character at a time:

```
fid = fopen('bench.dat');

k = 0;
while ~feof(fid)
    curr = fscanf(fid,'%c',1);
    if ~isempty(curr)
       k = k+1;
       benchstr(k) = curr;
    end
end

fclose(fid);
```

See Also

`fclose` | `ferror` | `fopen` | `frewind` | `fseek` | `ftell`

How To

- “Testing for End of File (EOF)”

Purpose

Information about file I/O errors

Syntax

```
message = ferror(fileID)
[message, errnum] = ferror(fileID)
[...] = ferror(fileID, 'clear')
```

Description

`message = ferror(fileID)` returns the error message for the most recent file I/O operation on the specified file. If the operation was successful, *message* is an empty string. *fileID* is an integer file identifier obtained from `fopen`, or an identifier reserved for standard input (0), standard output (1), or standard error (2).

`[message, errnum] = ferror(fileID)` returns the error number. If the most recent file I/O operation was successful, *errnum* is 0. Negative error numbers correspond to MATLAB error messages. Positive error numbers correspond to C library error messages for your system.

`[...] = ferror(fileID, 'clear')` clears the error indicator for the specified file.

See Also

`fclose` | `fopen` | `fseek` | `ftell` | `feof` | `fscanf` | `fprintf` | `fread` | `fwrite`

feval

Purpose

Evaluate function

Syntax

```
[y1, y2, ...] = feval(fhandle, x1, ..., xn)
[y1, y2, ...] = feval(fname, x1, ..., xn)
```

Description

`[y1, y2, ...] = feval(fhandle, x1, ..., xn)` evaluates the function handle, `fhandle`, using arguments `x1` through `xn`. If the function handle is bound to more than one built-in or `.m` function, (that is, it represents a set of overloaded functions), then the data type of the arguments `x1` through `xn` determines which function is dispatched to.

Note It is not necessary to use `feval` to call a function by means of a function handle. This is explained in "Calling a Function Using Its Handle" in the MATLAB Programming Fundamentals documentation.

`[y1, y2, ...] = feval(fname, x1, ..., xn)`. If `fname` is a quoted string containing the name of a function (usually defined within file having a .m file extension), then `feval(fname, x1, ..., xn)` evaluates that function at the given arguments. The `fname` parameter must be a simple function name; it cannot contain path information.

Tips

The following two statements are equivalent.

```
[V,D] = eig(A)
[V,D] = feval(@eig, A)
```

Nested functions are not accessible to `feval`. To call a nested function, you must either call it directly by name, or construct a function handle for it using the `@` operator.

Examples

The following example passes a function handle, `fhandle`, in a call to `fminbnd`. The `fhandle` argument is a handle to the `humps` function.

```
fhandle = @humps;
x = fminbnd(fhandle, 0.3, 1);
```

The `fminbnd` function uses `feval` to evaluate the function handle that was passed in.

```
function [xf, fval, exitflag, output] = ...
    fminbnd(funfcn, ax, bx, options, varargin)
            .
            .
            .
fx = feval(funfcn, x, varargin{:});
```

See Also

`assignin` | `function_handle` | `functions` | `builtin` | `eval` | `evalin`

Feval (COM)

Purpose

Evaluate MATLAB function in Automation server

Syntax

MATLAB Client

```
result = h.Feval('functionname',numout,arg1,arg2,...)
result = Feval(h,'functionname',numout,arg1,arg2,...)
result = invoke(h,'Feval','functionname',numout,...
arg1,arg2,...)
```

IDL Method Signature

```
HRESULT Feval([in] BSTR functionname, [in] long nargout,
[out] VARIANT* result, [in, optional] VARIANT arg1, arg2, ...)
```

Microsoft Visual Basic Client

```
Feval(String functionname, long numout,
arg1, arg2, ...) As Object
```

Description

`Feval` executes the MATLAB function specified by the string `functionname` in the Automation server attached to handle `h`.

Indicate the number of outputs to be returned by the function in a `1`-by-`1` `double` array, `numout`. The server returns output from the function in the cell array, `result`.

You can specify as many as 32 input arguments to be passed to the function. These arguments follow `numout` in the `Feval` argument list. The following table shows ways to pass an argument.

Passing Mechanism	Description
Pass the value itself	To pass any numeric or string value, specify the value in the `Feval` argument list: `a = h.Feval('sin', 1, -pi:0.01:pi);`
Pass a client variable	To pass an argument assigned to a variable in the client, specify the variable name alone: `x = -pi:0.01:pi;`

Passing Mechanism	Description
	`a = h.Feval('sin', 1, x);`

Reference a server variable	To reference a variable defined in the server, specify the variable name followed by an equals (=) sign:
	
	`h.PutWorkspaceData('x', 'base', -pi:0.01:pi);`
	`a = h.Feval('sin', 1, 'x=');`
	
	MATLAB does not reassign the server variable.

Tips

To display the output from `Feval` in the client window, assign a return value.

Server function names, like `Feval`, are case sensitive when using the first two syntaxes shown in the Syntax section.

COM functions are available on Microsoft Windows systems only.

Examples

Passing Arguments — MATLAB Client

This section contains a number of examples showing how to use `Feval` to execute MATLAB commands on a MATLAB Automation server.

- Concatenate two strings in the server by passing the input strings in a call to `strcat` through `Feval` (`strcat` deletes trailing spaces; use leading spaces):

  ```
  h = actxserver('matlab.application');
  a = h.Feval('strcat', 1, 'hello', ' world')
  ```

  MATLAB displays:

  ```
  a =
      'hello world'
  ```

- Perform the same concatenation, passing a string and a local variable `clistr` that contains the second string:

```
clistr = ' world';
a = h.Feval('strcat', 1, 'hello', clistr)
```

MATLAB displays:

```
a =
    'hello world'
```

- In this example, the variable `srvstr` is defined in the server, not the client. Putting an equals sign after a variable name (for example, `srvstr=`) indicates it is a server variable, and the variable is not defined in the client:

```
% Define the variable srvstr on the server.
h.PutCharArray('srvstr', 'base', ' world')

% Pass the name of the server variable using 'name=' syntax
a = h.Feval('strcat', 1, 'hello', 'srvstr=')
```

MATLAB displays:

```
a =
    'hello world'
```

Visual Basic .NET Client

Here are the same examples shown above, but written for a Visual Basic .NET client. These examples return the same strings as shown above.

- Pass the two strings to the MATLAB function `strcat` on the server:

```
Dim Matlab As Object
Dim out As Object
out = Nothing
Matlab = CreateObject("matlab.application")
Matlab.Feval("strcat", 1, out, "hello", " world")
```

- Define `clistr` locally and pass this variable:

```
Dim clistr As String
```

```
clistr = " world"
Matlab.Feval("strcat", 1, out, "hello", clistr)
```

- Pass the name of a variable defined on the server:

```
Matlab.PutCharArray("srvstr", "base", " world")
Matlab.Feval("strcat", 1, out, "hello", "srvstr=")
```

Feval Return Values — MATLAB Client. `Feval` returns data from the evaluated function in a cell array. The cell array has one row for every return value. You control the number of return values using the `numout` argument.

The `numout` argument in the following example specifies that `Feval` return three outputs from the `fileparts` function.

```
a = h.Feval('fileparts', 3, 'd:\work\ConsoleApp.cpp')
```

MATLAB displays:

```
a =
    'd:\work'
    'ConsoleApp'
    '.cpp'
```

Convert the returned values from the cell array `a` to char arrays:

```
a{:}
```

MATLAB displays:

```
ans =
d:\work

ans =
ConsoleApp

ans =
.cpp
```

Feval Return Values — Visual Basic .NET Client

Here is the same example, but coded in Visual Basic. Define the argument returned by `Feval` as an `Object`.

```
Dim Matlab As Object
Dim out As Object
Matlab = CreateObject("matlab.application")
Matlab.Feval("fileparts", 3, out, "d:\work\ConsoleApp.cpp")
```

See Also

`Execute` | `PutFullMatrix` | `GetFullMatrix` | `PutCharArray` | `GetCharArray`

Purpose

Fast Fourier transform

Syntax

```
Y = fft(x)
Y = fft(X,n)
Y = fft(X,[],dim)
Y = fft(X,n,dim)
```

Definitions

The functions `Y = fft(x)` and `y = ifft(X)` implement the transform and inverse transform pair given for vectors of length N by:

$$X(k) = \sum_{j=1}^{N} x(j)\omega_N^{(j-1)(k-1)}$$

$$x(j) = (1/N)\sum_{k=1}^{N} X(k)\omega_N^{-(j-1)(k-1)}$$

where

$$\omega_N = e^{(-2\pi i)/N}$$

is an Nth root of unity.

Description

`Y = fft(x)` returns the discrete Fourier transform (DFT) of vector `x`, computed with a fast Fourier transform (FFT) algorithm.

If the input `X` is a matrix, `Y = fft(X)` returns the Fourier transform of each column of the matrix.

If the input `X` is a multidimensional array, `fft` operates on the first nonsingleton dimension.

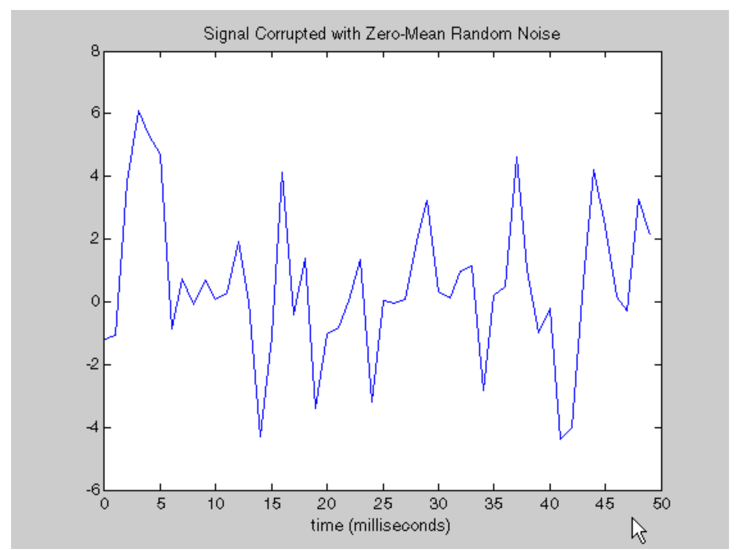
`Y = fft(X,n)` returns the `n`-point DFT. `fft(X)` is equivalent to `fft(X, n)` where `n` is the size of `X` in the first nonsingleton dimension. If the length of `X` is less than `n`, `X` is padded with trailing zeros to length `n`. If the length of `X` is greater than `n`, the sequence `X` is truncated. When `X` is a matrix, the length of the columns are adjusted in the same manner.

`Y = fft(X,[],dim)` and `Y = fft(X,n,dim)` applies the FFT operation across the dimension `dim`.

Examples

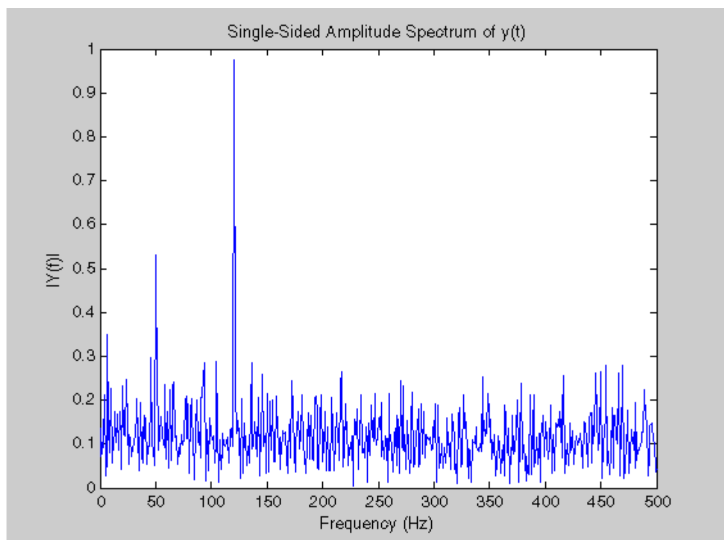
A common use of Fourier transforms is to find the frequency components of a signal buried in a noisy time domain signal. Consider data sampled at 1000 Hz. Form a signal containing a 50 Hz sinusoid of amplitude 0.7 and 120 Hz sinusoid of amplitude 1 and corrupt it with some zero-mean random noise:

```
Fs = 1000;                    % Sampling frequency
T = 1/Fs;                     % Sample time
L = 1000;                     % Length of signal
t = (0:L-1)*T;                % Time vector
% Sum of a 50 Hz sinusoid and a 120 Hz sinusoid
x = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t);
y = x + 2*randn(size(t));     % Sinusoids plus noise
plot(Fs*t(1:50),y(1:50))
title('Signal Corrupted with Zero-Mean Random Noise')
xlabel('time (milliseconds)')
```



It is difficult to identify the frequency components by looking at the original signal. Converting to the frequency domain, the discrete Fourier transform of the noisy signal `y` is found by taking the fast Fourier transform (FFT):

```
NFFT = 2^nextpow2(L); % Next power of 2 from length of y
Y = fft(y,NFFT)/L;
f = Fs/2*linspace(0,1,NFFT/2+1);

% Plot single-sided amplitude spectrum.
plot(f,2*abs(Y(1:NFFT/2+1)))
title('Single-Sided Amplitude Spectrum of y(t)')
xlabel('Frequency (Hz)')
ylabel('|Y(f)|')
```



The main reason the amplitudes are not exactly at 0.7 and 1 is because of the noise. Several executions of this code (including recomputation of `y`) will produce different approximations to 0.7 and 1. The other reason is that you have a finite length signal. Increasing `L` from 1000 to

10000 in the example above will produce much better approximations on average.

Algorithms

The FFT functions (`fft`, `fft2`, `fftn`, `ifft`, `ifft2`, `ifftn`) are based on a library called FFTW [3],[4]. To compute an N-point DFT when N is composite (that is, when $N = N_1N_2$), the FFTW library decomposes the problem using the Cooley-Tukey algorithm [1], which first computes N_1 transforms of size N_2, and then computes N_2 transforms of size N_1. The decomposition is applied recursively to both the N_1- and N_2-point DFTs until the problem can be solved using one of several machine-generated fixed-size "codelets." The codelets in turn use several algorithms in combination, including a variation of Cooley-Tukey [5], a prime factor algorithm [6], and a split-radix algorithm [2]. The particular factorization of N is chosen heuristically.

When N is a prime number, the FFTW library first decomposes an N-point problem into three $(N - 1)$-point problems using Rader's algorithm [7]. It then uses the Cooley-Tukey decomposition described above to compute the $(N - 1)$-point DFTs.

For most N, real-input DFTs require roughly half the computation time of complex-input DFTs. However, when N has large prime factors, there is little or no speed difference.

The execution time for `fft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `fft` using the utility function `fftw`, which controls the optimization of the algorithm used to compute an FFT of a particular size and dimension.

Data Type Support

`fft` supports inputs of data types `double` and `single`. If you call `fft` with the syntax `y = fft(X, ...)`, the output `y` has the same data type as the input `X`.

References

[1] Cooley, J. W. and J. W. Tukey, "An Algorithm for the Machine Computation of the Complex Fourier Series,"*Mathematics of Computation*, Vol. 19, April 1965, pp. 297-301.

[2] Duhamel, P. and M. Vetterli, "Fast Fourier Transforms: A Tutorial Review and a State of the Art," *Signal Processing*, Vol. 19, April 1990, pp. 259-299.

[3] FFTW (`http://www.fftw.org`)

[4] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT,"*Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

[5] Oppenheim, A. V. and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, p. 611.

[6] Oppenheim, A. V. and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, p. 619.

[7] Rader, C. M., "Discrete Fourier Transforms when the Number of Data Samples Is Prime," *Proceedings of the IEEE*, Vol. 56, June 1968, pp. 1107-1108.

See Also

`fft2` | `fftn` | `fftw` | `fftshift` | `ifft` | `dftmtx` | `filter` | `freqz`

Purpose

2-D fast Fourier transform

Syntax

```
Y = fft2(X)
Y = fft2(X,m,n)
```

Description

`Y = fft2(X)` returns the two-dimensional discrete Fourier transform (DFT) of `X`. The DFT is computed with a fast Fourier transform (FFT) algorithm. The result, `Y`, is the same size as `X`.

If the dimensionality of `X` is greater than `2`, the `fft2` function returns the 2-D DFT for each higher dimensional slice of `X`. For example, if `size(X) = [100 100 3]`, then `fft2` computes the DFT of `X(:,:,1)`, `X(:,:,2)` and `X(:,:,3)`.

`Y = fft2(X,m,n)` truncates `X`, or pads `X` with zeros to create an `m`-by-`n` array before doing the transform. The result is `m`-by-`n`.

Algorithms

`fft2(X)` can be simply computed as

```
fft(fft(X).').'
```

This computes the one-dimensional DFT of each column `X`, then of each row of the result. The execution time for `fft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `fft2` using the utility function `fftw`, which controls how MATLAB software optimizes the algorithm used to compute an FFT of a particular size and dimension.

Data Type Support

`fft2` supports inputs of data types `double` and `single`. If you call `fft2` with the syntax `y = fft2(X, ...)`, the output `y` has the same data type as the input `X`.

See Also

fft | fftn | fftw | fftshift | ifft2

fftn

Purpose

N-D fast Fourier transform

Syntax

```
Y = fftn(X)
Y = fftn(X,siz)
```

Description

`Y = fftn(X)` returns the discrete Fourier transform (DFT) of `X`, computed with a multidimensional fast Fourier transform (FFT) algorithm. The result `Y` is the same size as `X`.

`Y = fftn(X,siz)` pads `X` with zeros, or truncates `X`, to create a multidimensional array of size `siz` before performing the transform. The size of the result `Y` is `siz`.

Algorithms

`fftn(X)` is equivalent to

```
Y = X;
for p = 1:length(size(X))
    Y = fft(Y,[],p);
end
```

This computes in-place the one-dimensional fast Fourier transform along each dimension of `X`. The execution time for `fft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `fftn` using the utility function `fftw`, which controls the optimization of the algorithm used to compute an FFT of a particular size and dimension.

Data Type Support

`fftn` supports inputs of data types `double` and `single`. If you call `fftn` with the syntax `y = fftn(X, ...)`, the output `y` has the same data type as the input `X`.

See Also

`fft` | `fft2` | `fftn` | `fftw` | `ifftn`

fftshift

Purpose

Shift zero-frequency component to center of spectrum

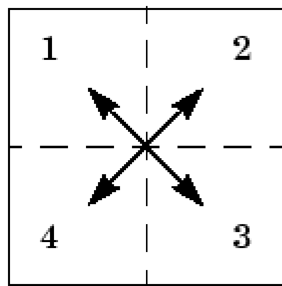
Syntax

```
Y = fftshift(X)
Y = fftshift(X,dim)
```

Description

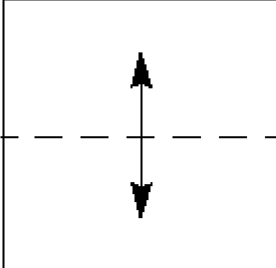
`Y = fftshift(X)` rearranges the outputs of `fft`, `fft2`, and `fftn` by moving the zero-frequency component to the center of the array. It is useful for visualizing a Fourier transform with the zero-frequency component in the middle of the spectrum.

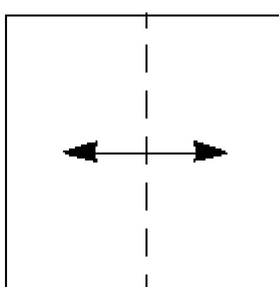
For vectors, `fftshift(X)` swaps the left and right halves of `X`. For matrices, `fftshift(X)` swaps the first quadrant with the third and the second quadrant with the fourth.



For higher-dimensional arrays, `fftshift(X)` swaps “half-spaces” of `X` along each dimension.

`Y = fftshift(X,dim)` applies the `fftshift` operation along the dimension `dim`.





Note `ifftshift` will undo the results of `fftshift`. If the matrix `X` contains an odd number of elements, `ifftshift(fftshift(X))` must be done to obtain the original `X`. Simply performing `fftshift(X)` twice will not produce `X`.

Examples

For any matrix `X`

```
Y = fft2(X)
```

has `Y(1,1) = sum(sum(X))`; the zero-frequency component of the signal is in the upper-left corner of the two-dimensional FFT. For

```
Z = fftshift(Y)
```

this zero-frequency component is near the center of the matrix.

The difference between `fftshift` and `ifftshift` is important for input sequences of odd-length.

```
N = 5;
X = 0:N-1;
Y = fftshift(fftshift(X));
Z = ifftshift(fftshift(X));
```

Notice that `Z` is a correct replica of `X`, but `Y` is not.

```
isequal(X,Y),isequal(X,Z)

ans =

     0

ans =

     1
```

See Also

circshift | fft | fft2 | fftn | ifftshift

Purpose

Interface to FFTW library run-time algorithm tuning control

Syntax

```
fftw('planner', method)
method = fftw('planner')
str = fftw('dwisdom')
str = fftw('swisdom')
fftw('dwisdom', str)
fftw('swisdom', str)
```

Description

`fftw` enables you to optimize the speed of the MATLAB FFT functions `fft`, `ifft`, `fft2`, `ifft2`, `fftn`, and `ifftn`. You can use `fftw` to set options for a tuning algorithm that experimentally determines the fastest algorithm for computing an FFT of a particular size and dimension at run time. MATLAB software records the optimal algorithm in an internal data base and uses it to compute FFTs of the same size throughout the current session. The tuning algorithm is part of the FFTW library that MATLAB software uses to compute FFTs.

`fftw('planner', method)` sets the method by which the tuning algorithm searches for a good FFT algorithm when the dimension of the FFT is not a power of 2. You can specify `method` to be one of the following. The default method is `estimate`:

- `'estimate'`
- `'measure'`
- `'patient'`
- `'exhaustive'`
- `'hybrid'`

When you call `fftw('planner', method)`, the next time you call one of the FFT functions, such as `fft`, the tuning algorithm uses the specified method to optimize the FFT computation. Because the tuning involves trying different algorithms, the first time you call an FFT function, it might run more slowly than if you did not call `fftw`. However, subsequent calls to any of the FFT functions, for a problem of the same size, often run more quickly than they would without using `fftw`.

Note The FFT functions only use the optimal FFT algorithm during the current MATLAB session. "Reusing Optimal FFT Algorithms" on page 1-1630 explains how to reuse the optimal algorithm in a future MATLAB session.

If you set the method to `'estimate'`, the FFTW library does not use run-time tuning to select the algorithms. The resulting algorithms might not be optimal.

If you set the method to `'measure'`, the FFTW library experiments with many different algorithms to compute an FFT of a given size and chooses the fastest. Setting the method to `'patient'` or `'exhaustive'` has a similar result, but the library experiments with even more algorithms so that the tuning takes longer the first time you call an FFT function. However, subsequent calls to FFT functions are faster than with `'measure'`.

If you set `'planner'` to `'hybrid'`, MATLAB software

- Sets method to `'measure'` method for FFT dimensions 8192 or smaller.
- Sets method to `'estimate'` for FFT dimensions greater than 8192.

`method = fftw('planner')` returns the current planner method.

`str = fftw('dwisdom')` returns the information in the FFTW library's internal double-precision database as a string. The string can be saved and then later reused in a subsequent MATLAB session using the next syntax.

`str = fftw('swisdom')` returns the information in the FFTW library's internal single-precision database as a string.

`fftw('dwisdom', str)` loads `fftw` wisdom represented by the string `str` into the FFTW library's internal double-precision wisdom database. `fftw('dwisdom','')` or `fftw('dwisdom',[])` clears the internal wisdom database.

`fftw('swisdom', str)` loads `fftw` wisdom represented by the string `str` into the FFTW library's internal single-precision wisdom database. `fftw('swisdom','')` or `fftw('swisdom',[])` clears the internal wisdom database.

Note on large powers of 2 For FFT dimensions that are powers of 2, between 2^{14} and 2^{22}, MATLAB software uses special preloaded information in its internal database to optimize the FFT computation. No tuning is performed when the dimension of the FTT is a power of 2, unless you clear the database using the command `fftw('wisdom', [])`.

For more information about the FFTW library, see `http://www.fftw.org`.

Examples

Comparison of Speed for Different Planner Methods

The following example illustrates the run times for different settings of `planner`. The example first creates some data and applies `fft` to it using the default method, `estimate`.

```
t=0:.001:5;
x = sin(2*pi*50*t)+sin(2*pi*120*t);
y = x + 2*randn(size(t));

tic; Y = fft(y,1458); toc
Elapsed time is 0.000521 seconds.
```

If you execute the commands

```
tic; Y = fft(y,1458); toc
Elapsed time is 0.000151 seconds.
```

a second time, MATLAB software reports the elapsed time as essentially 0. To measure the elapsed time more accurately, you can execute the command `Y = fft(y,1458)` 1000 times in a loop.

```
tic; for k=1:1000
Y = fft(y,1458);
end; toc
Elapsed time is 0.056532 seconds.
```

This tells you that it takes on order of 1/10000 of a second to execute `fft(y, 1458)` a single time.

For comparison, set `planner` to `patient`. Since this `planner` explores possible algorithms more thoroughly than `hybrid`, the first time you run `fft`, it takes longer to compute the results.

```
fftw('planner','patient')
tic;Y = fft(y,1458);toc
Elapsed time is 0.100637 seconds.
```

However, the next time you call `fft`, it runs at approximately the same speed as before you ran the method `patient`.

```
tic;for k=1:1000
Y=fft(y,1458);
end;toc
Elapsed time is 0.057209 seconds.
```

Reusing Optimal FFT Algorithms

In order to use the optimized FFT algorithm in a future MATLAB session, first save the “wisdom” using the command

```
str = fftw('wisdom')
```

You can save `str` for a future session using the command

```
save str
```

The next time you open a MATLAB session, load `str` using the command

```
load str
```

and then reload the "wisdom" into the FFTW database using the command

```
fftw('wisdom', str)
```

See Also

`fft` | `fft2` | `fftn` | `ifft` | `ifft2` | `ifftn` | `fftshift`

Purpose

Read line from file, removing newline characters

Syntax

`tline` `= fgetl(`*`fileID`*`)`

Description

`tline` `= fgetl(`*`fileID`*`)` returns the next line of the specified file, removing the newline characters. *`fileID`* is an integer file identifier obtained from `fopen`. *`tline`* is a text string unless the line contains only the end-of-file marker. In this case, *`tline`* is the numeric value `-1`.

`fgetl` reads characters using the encoding scheme associated with the file. To specify the encoding scheme, use `fopen`.

Examples

Read and display the file `fgetl.m` one line at a time:

```
fid = fopen('fgetl.m');

tline = fgetl(fid);
while ischar(tline)
    disp(tline)
    tline = fgetl(fid);
end

fclose(fid);
```

Compare these results to the `fgets` example, which replaces the calls to `fgetl` with `fgets`.

See Also

`fclose` | `feof` | `ferror` | `fgets` | `fopen` | `fprintf` | `fread` | `fscanf` | `fwrite`

How To

- "Testing for EOF with fgetl and fgets"

Purpose

Read line of text from device and discard terminator

Syntax

```
tline = fgetl(obj)
[tline,count] = fgetl(obj)
[tline,count,msg] = fgetl(obj)
```

Description

`tline = fgetl(obj)` reads one line of text from the device connected to the serial port object, `obj`, and returns the data to `tline`. This returned data does not include the terminator with the text line. To include the terminator, use `fgets`.

`[tline,count] = fgetl(obj)` returns the number of values read to `count`, including the terminator.

`[tline,count,msg] = fgetl(obj)` returns a warning message to `msg` if the read operation was unsuccessful.

Tips

Before you can read text from the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read – including the terminator – each time `fgetl` is issued.

If you use the `help` command to display help for `fgetl`, then you need to supply the pathname shown below.

```
help serial/fgetl
```

Rules for Completing a Read Operation with fgetl

A read operation with `fgetl` blocks access to the MATLAB command line until:

- The terminator specified by the `Terminator` property is reached.

- The time specified by the `Timeout` property passes.
- The input buffer is filled.

Examples

On a Windows platform, create the serial port object `s`, connect `s` to a Tektronix® TDS 210 oscilloscope, and write the `RS232?` command with the `fprintf` function. `RS232?` instructs the scope to return serial port communications settings.

```
s = serial('COM1');
fopen(s)
fprintf(s,'RS232?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data is automatically returned to the input buffer.

```
s.BytesAvailable
ans =
    17
```

Use `fgetl` to read the data returned from the previous write operation, and discard the terminator.

```
settings = fgetl(s)
settings =
9600;0;0;NONE;LF
length(settings)
ans =
    16
```

Disconnect `s` from the scope, and remove `s` from memory and the workspace.

```
fclose(s)
delete(s)
clear s
```

See Also

`fgets` | `fopen` | `BytesAvailable` | `InputBufferSize` | `ReadAsyncMode` | `Status` | `Terminator` | `Timeout` | `ValuesReceived`

fgets

Purpose

Read line from file, keeping newline characters

Syntax

```
tline = fgets(fileID)
tline = fgets(fileID, nchar)
```

Description

tline = fgets(*fileID*) reads the next line of the specified file, including the newline characters. *fileID* is an integer file identifier obtained from fopen. *tline* is a text string unless the line contains only the end-of-file marker. In this case, *tline* is the numeric value -1. fgets reads characters using the encoding scheme associated with the file. To specify the encoding scheme, use fopen.

tline = fgets(*fileID*, *nchar*) returns at most *nchar* characters of the next line. *tline* does not include any characters after the newline characters or the end-of-file marker.

Examples

Read and display the file fgets.m. Because fgets keeps newline characters and disp adds a newline character, this code displays the file with double-spacing:

```
fid = fopen('fgets.m');

tline = fgets(fid);
while ischar(tline)
    disp(tline)
    tline = fgets(fid);
end

fclose(fid);
```

Compare these results to the fgetl example, which replaces the calls to fgets with fgetl.

See Also

fclose | feof | ferror | fgetl | fopen | fprintf | fread | fscanf | fwrite

How To

- "Testing for EOF with fgetl and fgets"

Purpose

Read line of text from device and include terminator

Syntax

```
tline = fgets(obj)
[tline,count] = fgets(obj)
[tline,count,msg] = fgets(obj)
```

Description

`tline = fgets(obj)` reads one line of text from the device connected to the serial port object, `obj`, and returns the data to `tline`. This returned data includes the terminator with the text line. To exclude the terminator, use `fgetl`.

`[tline,count] = fgets(obj)` returns the number of values read to `count`, including the terminator.

`[tline,count,msg] = fgets(obj)` returns a warning message to `msg` if the read operation was unsuccessful.

Tips

Before you can read text from the device, it must be connected to `obj` with the `fopen`function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read – including the terminator – each time `fgets` is issued.

If you use the `help` command to display help for `fgets`, then you need to supply the pathname shown below.

```
help serial/fgets
```

Rules for Completing a Read Operation with fgets

A read operation with `fgets` blocks access to the MATLAB command line until:

- The terminator specified by the `Terminator` property is reached.

- The time specified by the `Timeout` property passes.
- The input buffer is filled.

Examples

Create the serial port object `s`, connect `s` to a Tektronix TDS 210 oscilloscope, and write the `RS232?` command with the `fprintf` function. `RS232?` instructs the scope to return serial port communications settings.

```
s = serial('COM1');
fopen(s)
fprintf(s,'RS232?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data is automatically returned to the input buffer.

```
s.BytesAvailable
ans =
    17
```

Use `fgets` to read the data returned from the previous write operation, and include the terminator.

```
settings = fgets(s)
settings =
9600;0;0;NONE;LF
length(settings)
ans =
    17
```

Disconnect `s` from the scope, and remove `s` from memory and the workspace.

```
fclose(s)
delete(s)
clear s
```

See Also

fgetl | fopen | BytesAvailable | BytesAvailableFcn | InputBufferSize | Status | Terminator | Timeout | ValuesReceived

fieldnames

Purpose

Field names of structure, or public fields of object

Syntax

```
names = fieldnames(s)
names = fieldnames(obj)
names = fieldnames(obj,'-full')
```

Description

`names = fieldnames(s)` returns a cell array of strings containing the names of the fields in structure `s`.

`names = fieldnames(obj)` returns a cell array of strings containing the names of the public properties of `obj`. MATLAB objects can overload `fieldnames` and define their own behavior.

`names = fieldnames(obj,'-full')` returns a cell array of strings containing the name, type, attributes, and inheritance of the properties of `obj`. Only supported for COM or Java objects.

Examples

Structure Fields

Create a structure array and view its fields.

```
s(1,1).name = 'alice';
s(1,1).ID = 0;
s(2,1).name = 'gertrude';
s(2,1).ID = 1;

names = fieldnames(s)
```

```
names =
    'name'
    'ID'
```

Java Object Properties

Create a Java object and view its public properties.

```
obj = java.lang.Integer(0);
names = fieldnames(obj)
```

```
names =
    'MIN_VALUE'
    'MAX_VALUE'
    'TYPE'
    'SIZE'
```

See Also

setfield | getfield | isfield | orderfields | rmfield

How To

- “Generate Field Names from Variables”

figure

Purpose

Create figure graphics object

Syntax

```
figure
figure('PropertyName',propertyvalue,...)
figure(h)
h = figure(...)
```

Properties

For a list of properties, see `Figure Properties`.

Description

`figure` creates figure graphics objects. Figure objects are the individual windows on the screen in which the MATLAB software displays graphical output.

`figure` creates a new figure object using default property values. This automatically becomes the current figure and raises it above all other figures on the screen until a new figure is either created or called.

`figure('PropertyName',propertyvalue,...)` creates a new figure object using the values of the properties specified. For a description of the properties, see `Figure Properties`. MATLAB uses default values for any properties that you do not explicitly define as arguments.

`figure(h)` does one of two things, depending on whether or not a figure with handle `h` exists. If `h` is the handle to an existing figure, `figure(h)` makes the figure identified by `h` the current figure, makes it visible, and raises it above all other figures on the screen. The current figure is the target for graphics output. If `h` is not the handle to an existing figure, but is an integer, `figure(h)` creates a figure and assigns it the handle `h`. `figure(h)` where `h` is not the handle to a figure, and is not an integer, is an error.

`h = figure(...)` returns the handle to the figure object.

Tips

To create a figure object, MATLAB creates a new window whose characteristics are controlled by default figure properties (both factory installed and user defined) and properties specified as arguments. See `Figure Properties` for a description of these properties.

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the `set` and `get` reference pages for examples of how to specify these data types).

Use `set` to modify the properties of an existing figure or `get` to query the current values of figure properties.

The `gcf` command returns the handle to the current figure and is useful as an argument to the `set` and `get` commands.

Figures can be docked in the desktop. The `DockControls` property determines whether you can dock the figure.

Making a Figure Current

The current figure is the target for graphics output. There are two ways to make a figure `h` the current figure.

- Make the figure `h` current, visible, and displayed on top of other figures:

  ```
  figure(h)
  ```

- Make the figure h current, but do not change its visibility or stacking with respect to other figures:

  ```
  set(0,'CurrentFigure',h)
  ```

Examples

Specifying Figure Size and Screen Location

To create a figure window that is one quarter the size of your screen and is positioned in the upper left corner, use the root object's `ScreenSize` property to determine the size. `ScreenSize` is a four-element vector: `[left, bottom, width, height]`:

```
scrsz = get(0,'ScreenSize');
figure('Position',[1 scrsz(4)/2 scrsz(3)/2 scrsz(4)/2])
```

To position the full figure window including the menu bar, title bar, tool bars, and outer edges, use the `OuterPosition` property in the same manner.

Specifying the Figure Window Title

You can add your own title to a figure by setting the `Name` property and you can turn off the figure number with the `NumberTitle` property:

```
figure('Name','Simulation Plot Window','NumberTitle','off')
```

See `Figure Properties` for a description of all properties.

Setting Default Properties

You can set default figure properties only on the `root object` level.

```
set(0,'DefaultFigureProperty',PropertyValue...)
```

where *`Property`* is the name of the figure property and `PropertyValue` is the value you are specifying. Use `set` and `get` to access figure properties.

See "Setting Default Property Values" for more information.

See Also

`axes` | `close` | `clf` | `gcf` | `ishghandle` | `rootobject` | `uicontrol` | `uimenu` | `Figure Properties`

Figure Properties

Purpose

Define figure properties

Creating Figure Objects

Use `figure` to create figure objects.

Modifying Properties

You can set and query graphics object properties in two ways:

- "The Property Editor" is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of Handle Graphics properties.

To change the default values of properties, see "Setting Default Property Values" in the Handle Graphics Objects documentation.

Figure Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`Alphamap`

`m-by-1` matrix of alpha values

Figure alphamap. An array of non-NaN alpha values. MATLAB accesses alpha values by their row number. For example, an index of `1` specifies the first alpha value, an index of `2` specifies the second alpha value, and so on. Alphamaps can be any length. The default alphamap contains 64 values that progress linearly from `0` to `1`.

Alphamaps affect the rendering of surface, image, and patch objects, but do not affect other graphics objects.

`BeingDeleted`

`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted`

property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

See the `close` and `delete` function reference pages for related information.

`BusyAction`

`cancel | {queue}`

Callback function interruption. The `BusyAction` property enables you to control how MATLAB handles events that potentially interrupt executing callback functions. If there is a callback function executing, callback functions invoked subsequently always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is `on` (the default), then interruption occurs at the next point where the event queue is processed. If the `Interruptible` property is `off`, the `BusyAction` property (of the object owning the executing callback) determines how MATLAB handles the event. The choices are

- `cancel` — Discard the event that attempted to execute a second callback function.
- `queue` — Queue the event that attempted to execute a second callback function until the current callback finishes.

`ButtonDownFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Button press callback function. Executes whenever you press a mouse button while the pointer is in the figure window, but not over a child object (i.e., uicontrol, uipanel, axes, or axes child).

Define the `ButtonDownFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with the mouse button press and an empty event structure).

See the figure's `SelectionType` property to determine whether modifier keys were also pressed.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

Using the ButtonDownFcn

This example creates a figure and defines a function handle callback for the `ButtonDownFcn` property. When the user **Ctrl-**clicks the figure, the callback creates a new figure having the same callback.

Click to view in editor — This link opens the MATLAB Editor with the following example.

Click to run example — **Ctrl-**click the figure to create a new figure.

```
fh_cb = @newfig; % Create function handle for newfig function
figure('ButtonDownFcn',fh_cb);

function newfig(src,evnt)
   if strcmp(get(src,'SelectionType'),'alt')
      figure('ButtonDownFcn',fh_cb)
   else
      disp('Use control-click to create a new figure')
   end
end
```

`Children`
vector of handles

Children of the figure. A vector containing the handles of all axes, user-interface objects displayed within the figure. You can change the order of the handles and thereby change the stacking of the objects on the display.

When an object's `HandleVisibility` property is `off`, it is not listed in its parent's `Children` property. See `HandleVisibility` for more information.

`Clipping`

`{on} | off`

Clipping mode. This property has no effect on figures.

`CloseRequestFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Function executed on figure close. Executes whenever you issue the `close` command (either a `close(figure_handle)` or a `close all`), when you close a figure window from the computer's window manager menu, or when you quit MATLAB.

The `CloseRequestFcn` provides a mechanism to intervene in the closing of a figure. It allows you to, for example, display a dialog box to ask a user to confirm or cancel the close operation or to prevent users from closing a figure that contains a GUI.

The basic mechanism is:

1. A user issues the `close` command from the command line, by closing the window from the computer's window manager menu, or by quitting MATLAB.
2. The close operation executes the function defined by the figure `CloseRequestFcn`. The default function is `closereq`.

   `closereq` unconditionally deletes the current figure, destroying the window. `closereq` takes advantage of the fact that the

`close` command makes each figure specified as arguments the current figure before calling its respective close request function.

Note that `closereq` honors the `ShowHiddenHandles` setting during figure deletion and will not delete hidden figures.

Redefining the CloseRequestFcn

Define the `CloseRequestFcn` as a function handle. For example:

```
set(gcf,'CloseRequestFcn',@my_closefcn)
```

Where `@my_closefcn` is a function handle referencing function `my_closefcn`.

Unless the close request function calls `delete` or `close`, MATLAB never closes the figure. (Note that you can always call `delete(`*`figure_handle`*`)` from the command line if you have created a window with a nondestructive close request function.)

A useful application of the close request function is to display a question dialog box asking the user to confirm the close operation. The following function illustrates how to do this.

Click to view in editor — This link opens the MATLAB editor with the following example.

Click to run example — **Ctrl-**click the figure to create a new figure.

```
function my_closereq(src,evnt)
% User-defined close request function
% to display a question dialog box
   selection = questdlg('Close This Figure?',...
      'Close Request Function',...
      'Yes','No','Yes');
   switch selection,
```

```
      case 'Yes',
         delete(gcf)
      case 'No'
      return
   end
end
```

Now create a figure using the `CloseRequestFcn`:

```
figure('CloseRequestFcn',@my_closereq)
```

To make this function your default close request function, set a default value on the root level.

```
set(0,'DefaultFigureCloseRequestFcn',@my_closereq)
```

MATLAB then uses this setting for the `CloseRequestFcn` of all subsequently created figures.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`Color`
: `ColorSpec`

Background color. Controls the figure window background color. You can specify a color using a three-element vector of RGB values or one of the MATLAB predefined names. See `ColorSpec` for more information.

`Colormap`
: `m`-by-`3` matrix of RGB values

Figure colormap. An array of red, green, and blue (RGB) intensity values that define `m` individual colors. MATLAB accesses colors by their row number. For example, an index of `1` specifies the first RGB triplet, an index of `2` specifies the second RGB triplet, and so on.

Number of Colors Allowed

Colormaps can be any length, but must be three columns wide. The default figure colormap contains 64 predefined colors.

Objects That Use Colormaps

Colormaps affect the rendering of `surface`, image, and `patch` objects, but generally do not affect other graphics objects. See `colormap` and `ColorSpec` for more information.

`CreateFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback function executed during figure creation. Executes when MATLAB creates a figure object. You must define this property as a default value on the root level. For example, the statement:

```
set(0,'DefaultFigureCreateFcn',@fig_create)
```

defines a default value on the root level that causes all figures created to execute the setup function `fig_create`, which is defined below:

```
function fig_create(src,evnt)
set(src,'Color',[.2 .1 .5],...
   'IntegerHandle','off',...
   'MenuBar','none',...
   'ToolBar','none')
end
```

MATLAB executes the create function after setting all properties for the figure. Setting this property on an existing figure object has no effect.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo` or the handle of the object generating the callback (the source of the event). For example, this,

```
f = figure('CreateFcn',@(o,e) keyboard)
K>> gcbo
```

and this each return 1:

```
f = figure('CreateFcn',@(o,e) keyboard)
K>> o
```

`CurrentAxes`
handle of current axes

Target axes in this figure. MATLAB sets this property to the handle of the figure's current axes (the handle returned by the `gca` command when this figure is the current figure). In all figures for which axes children exist, there is always a current axes. The current axes does not have to be the topmost axes, and setting an axes to be the `CurrentAxes` does not restack it above all other axes.

You can make an axes current using the `axes` and `set` commands. For example, `axes(`*`axes_handle`*`)` and `set(gcf,'CurrentAxes',`*`axes_handle`*`)` both make the axes identified by the handle *`axes_handle`* the current axes. In addition, `axes(`*`axes_handle`*`)` restacks the axes above all other axes in the figure.

If a figure contains no axes, `get(gcf,'CurrentAxes')` returns the empty matrix. Note that the `gca` function actually creates an axes if one does not exist.

`CurrentCharacter`
single character

Last key pressed. MATLAB sets this property to the last key pressed in the figure window. Use `CurrentCharacter` to obtain user input.

`CurrentObject`
object handle

Handle of current object. MATLAB sets this property to the handle of the last object clicked on by the mouse. This object is the frontmost object in the view. You can use this property to determine which object a user has selected. The function `gco` provides a convenient way to retrieve the `CurrentObject` of the `CurrentFigure`.

Note that the `HitTest` property controls whether an object can become the `CurrentObject`.

Hidden Handle Objects

Clicking an object whose `HandleVisibility` property is `off` (such as axis labels and title) causes the `CurrentObject` property to be set to empty `[]`. To avoid returning an empty value when users click hidden objects, set the hidden object's `HitTest` property to `off`.

Mouse Over

Note that cursor motion over objects does not update the `CurrentObject`; you must click objects to update this property. See the `CurrentPoint` property for related information.

`CurrentPoint`
two-element vector: [*x*-coordinate, *y*-coordinate]

Location of last button click in this figure. MATLAB sets this property to the location of the pointer at the time of the most recent mouse button press. MATLAB updates this property

whenever you press the mouse button while the pointer is in the figure window.

Note that if you select a point in the figure and then use the values returned by the CurrentPoint property to plot that point, there can be differences in the position due to round-off errors.

CurrentPoint and Cursor Motion

In addition to the behavior described above, MATLAB updates CurrentPoint before executing callback routines defined for the figure WindowButtonMotionFcn and WindowButtonUpFcn properties. This enables you to query CurrentPoint from these callback routines. It behaves like this:

- If there is no callback routine defined for the WindowButtonMotionFcn or the WindowButtonUpFcn, then MATLAB updates the CurrentPoint only when the mouse button is pressed down within the figure window.
- If there is a callback routine defined for the WindowButtonMotionFcn, then MATLAB updates the CurrentPoint just before executing the callback. Note that the WindowButtonMotionFcn executes only within the figure window unless the mouse button is pressed down within the window and then held down while the pointer is moved around the screen. In this case, the routine executes (and the CurrentPoint is updated) anywhere on the screen until the mouse button is released.
- If there is a callback routine defined for the WindowButtonUpFcn, MATLAB updates the CurrentPoint just before executing the callback. Note that the WindowButtonUpFcn executes only while the pointer is within the figure window unless the mouse button is pressed down initially within the window. In this case, releasing the button anywhere on the screen triggers callback execution, which is preceded by an update of the CurrentPoint.

The figure `CurrentPoint` is updated only when certain events occur, as previously described. In some situations (such as when the `WindowButtonMotionFcn` takes a long time to execute and the pointer is moved very rapidly), the `CurrentPoint` might not reflect the actual location of the pointer, but rather the location at the time when the `WindowButtonMotionFcn` began execution.

The `CurrentPoint` is measured from the lower-left corner of the figure window, in units determined by the `Units` property.

The root `PointerLocation` property contains the location of the pointer updated synchronously with pointer movement. However, the location is measured with respect to the screen, not a figure window.

See `uicontrol` for information on how this property is set when you click a uicontrol object.

`DeleteFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Delete figure callback function. Executes when the figure object is deleted (for example, when you issue a `delete` or a `close` command). MATLAB executes the function before destroying the object's properties so these values are available to the callback routine.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

The handle of the object whose `DeleteFcn` is being executed is accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See also the figure `CloseRequestFcn` property

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`DockControls`
`{on} | off`

Displays controls used to dock figure. Determines whether the figure enables the **Desktop** menu item and the dock figure button in the title bar that allow you to dock the figure into the MATLAB desktop.

- `on` — Figure docking controls are visible.
- `off` — The **Desktop** menu item that enables you to dock the figure is disabled and the figure dock button is not displayed.

See the `WindowStyle` property for more information on docking figure.

`DoubleBuffer`
`{on} | off`

Note This property is now obsolete and has no effect. It was provided for older computer systems to produce flash-free animations.

Flash-free rendering for simple animations. Double buffering is the process of drawing to an off-screen pixel buffer and then printing the buffer contents to the screen once the drawing is complete. Double buffering generally produces flash-free rendering for simple animations (such as those involving lines, as opposed to objects containing large numbers of polygons). Use double buffering with the animated objects' `EraseMode` property set to `normal`. Use the `set` command to disable double buffering.

```
set(figure_handle,'DoubleBuffer','off')
```

Double buffering works only when the figure `Renderer` property is `painters`.

`FileName`
String

GUI FIG-file name. GUIDE stores the name of the FIG-file used to save the GUI layout in this property. In non-GUIDE GUIs, by default `FileName` is empty. You can `set` the `FileName` property in non-GUIDE GUIs as well, and `get` it to verify what GUI is running or whether it has been previously saved.

`FixedColors`
`m`-by-`3` matrix of RGB values (read-only)

Noncolormap colors. Fixed colors define all colors appearing in a figure window that are not from the figure colormap. These colors include axis lines and labels, the colors of `line`, `text`, `uicontrol`, and `uimenu`, and `text` objects, and any colors explicitly defined, for example, with a statement like:

```
set(gcf,'Color',[0.3,0.7,0.9])
```

Fixed color definitions reside in the system color table and do not appear in the figure colormap. For this reason, fixed colors can limit the number of simultaneously displayed colors if the number of fixed colors plus the number of entries in the figure colormap exceed your system's maximum number of colors.

(See the root `ScreenDepth` property for information on determining the total number of colors supported on your system. See the `MinColorMap` property for information on how MATLAB shares colors between applications.)

Note The `FixedColors` property is being deprecated and will be removed in a future release

---

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

Handles are always visible when `HandleVisibility` is `on`.

- `on` — Handles are always visible.
- `callback` — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

Visibility and Handles Returned by Other Functions

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure`property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Making All Handles Visible

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Validity of Hidden Handles

Handles that are hidden are still valid. If you know an object's handle, you can pass it to any function that operates on handles, and set and get its properties.

`HitTest`
`{on} | off`

Selectable by mouse click. Determines if the figure can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the figure. If `HitTest` is `off`, clicking the figure sets the `CurrentObject` to the empty matrix.

`IntegerHandle`
`{on} | off`

Figure handle mode. Figure object handles are integers by default. When creating a new figure, MATLAB uses the lowest integer that is not used by an existing figure. If you delete a figure, its integer handle can be reused.

If you set this property to `off`, MATLAB assigns nonreusable real-number handles (for example, `67.0001221`) instead of integers. This feature is designed for dialog boxes where removing the handle from integer values reduces the likelihood of inadvertently drawing into the dialog box.

`Interruptible`
`{on} | off`

Callback routine interruption mode. Controls whether a figure callback function can be interrupted by subsequently invoked callbacks.

How Callbacks Are Interrupted

MATLAB checks for queued events that can interrupt a callback function only when it encounters a call to `drawnow`, `figure`, `getframe`, or `pause` in the executing callback function. When executing one of these functions, MATLAB processes all pending events, including executing all waiting callback functions. The interrupted callback then resumes execution.

What Property Callbacks Are Interruptible

The `Interruptible` property only affects callback functions defined for the `ButtonDownFcn`, `KeyPressFcn`, `KeyReleaseFcn`, `WindowButtonDownFcn`, `WindowButtonMotionFcn`, `WindowButtonUpFcn`, `WindowKeyPressFcn`, `WindowKeyReleaseFcn`, and `WindowScrollWheelFcn`.

See the `BusyAction` property for related information.

`InvertHardcopy`
`{on} | off`

Change hardcopy to black objects on white background. Affects only printed output. Printing a figure having a background color (`Color` property) that is not white results in poor contrast between graphics objects and the figure background and also consumes a lot of printer toner.

When `InvertHardCopy` is `on`, MATLAB eliminates this effect by changing the color of the figure and axes to white and the axis lines, tick marks, axis labels, etc., to black. `lines`, `text`, and the edges of `patch`es and `surface`s might be changed, depending on the `print` command options specified.

If you set `InvertHardCopy` to `off`, the printed output matches the colors displayed on the screen.

See `print` for more information on printing MATLAB figures.

`KeyPressFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Key press callback function. Callback function invoked by a key press that occurs while the figure window has focus. Define the `KeyPressFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with key press and an event structure).

See Function Handle Callbacks for information on how to use function handles to define the callback function.

When no callback is specified for this property (which is the default state), MATLAB passes any key presses to the Command Window. However, when you define a callback for this property, the figure retains focus with each key press and executes the specified callback with each key press.

KeyPressFcn Event Structure

When you specify the callback as a function handle, MATLAB passes to it a structure to containing the following fields.

Field	Contents
`Character`	The character displayed as a result of the pressing the key(s), which can be empty or unprintable
`Key`	The key being pressed, identified by the lowercase label on key or a descriptive string
`Modifier`	A cell array containing the names of one or more modifier keys being pressed (i.e., **control**, **alt**, **shift**). On Macintosh computers, it contains `'command'` when pressing the **command** modifier key

Explore KeyPressFcn Behavior

To view the values of the event data fields for any key or key combination, run the following code:

```
figure('NumberTitle','off','Menubar','none',...
       'Name','Press keys to put event data in Command Window',...
       'Position',[560 728 560 200],...
       'KeyPressFcn',@(obj,evt)disp(evt));
```

Each time you press a key, the `KeyPressFcn` uses `disp` to display the event data in the Command Window.

You can also view and run an example GUI, `ex_KeyPressFcn.m`, which displays keystroke event data in the figure window, and provides an option to write the event data structure to your workspace.

- Click here to view the example in the MATLAB editor
- Click here to add the example to the MATLAB path
- Click to run the example — Press and release various key combinations while the figure has focus. The callback displays event data in text fields in the figure window. The **Char**

Code data is the `Character` field displayed as a number, not a separate event data field.

Event data passed to a `KeyPressFcn` and `KeyReleaseFcn` callbacks have the following characteristics:

- The `Key` field is always in lower case (contains the non-shifted symbol).
- Modifier keys (**Alt**, **Ctrl**, **Shift**,) return data when pressed alone as well as when pressed in combination with other keys.
- `Modifier` fields contain a cell array with zero or more strings.
- Modifier keys can affect the `Character` field, but do not change the `Key` field.
- Certain keys, plus keys modified with **Ctrl**, put unprintable characters in the `Character` field.
- **Ctrl**, **Alt**, **Shift**, function and several other keys generate no `Character` field data.

Using the KeyPressFcn

This example creates a figure and defines a function handle callback for the `KeyPressFcn` property. When you press the **p** key, the callback exports the figure as a PNG image file. When you press **Ctrl+p**, the callback exports the figure as a PDF file.

```
function figure_keypress
    figure('KeyPressFcn',@printfig);
    surf(peaks)

    function printfig(src,event)
    % Callback to parse keypress event data to print a figure
        if event.Character == 'p'
            % On some systems you must send the file to a printer manually
            if length(event.Modifier) == 1 && ...
                strcmp(event.Modifier{:},'control')
```

```
                % Create PDF file of figure when Ctrl key is down
                print ('-dpdf',['-f' num2str(src)])
            elseif isempty(event.Modifier)
                % Print PNG image of figure when Ctrl is not pressed
                print ('-dpng','-r200',['-f' num2str(src)])
            end
        end
    end
end
```

`KeyReleaseFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Key release callback function. Callback function invoked by a key release that occurs while the figure window has focus. Define the `KeyReleaseFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with key release and an event structure).

See Function Handle Callbacks for information on how to use function handles to define the callback function.

KeyReleaseFcn Event Structure

When the callback is a function handle, MATLAB passes a structure as the second argument to the callback function that contains the following fields.

Field	Contents
`Character`	The character displayed as a result of the releasing the key(s), which can be empty or unprintable
`Key`	The key being released, identified by the lowercase label on key or a descriptive string
`Modifier`	A cell array containing the names of one or more modifier keys being released (i.e., **control**, **alt**, **shift**). On Macintosh computers, it contains `'command'` when releasing the **command** modifier key

Properties Affected by the KeyReleaseFcn

When a callback is defined for the `KeyReleaseFcn` property, MATLAB updates the `CurrentCharacter` figure property just before executing the callback.

Multiple-Key Press Events and a Single-Key Release Event

Consider a figure having callbacks defined for both the `KeyPressFcn` and `KeyReleaseFcn`. In the case where you press multiple keys at close to the same time, MATLAB generates repeated `KeyPressFcn` events only for the last key pressed.

For example, suppose you press and hold down the **a** key, then press and hold down the **s** key. MATLAB generates repeated `KeyPressFcn` events for the **a** key until you press the **s** key, at which point MATLAB generates repeated `KeyPressFcn` events for the **s** key. If you then release the **s** key, MATLAB generates a `KeyReleaseFcn` event for the **s** key, but no new `KeyPressFcn` events for the **a** key. When you then release the **a** key, the `KeyReleaseFcn` again executes. The `KeyReleaseFcn` executes its callback every time you release a key while the figure is in focus, regardless of what any `KeyPressFcn` does.

Event structures passed to a `KeyPressFcn` and `KeyReleaseFcn` callbacks have the following characteristics:

- The `Key` field is always in lower case (contains the non-shifted symbol).
- Modifier keys (**Alt**, **Ctrl**, **Shift**,) return data when pressed alone as well as when pressed in combination with other keys.
- `Modifier` fields contain a cell array with zero or more strings.
- Modifier keys can affect the `Character` field, but do not change the `Key` field.
- Certain keys, plus keys modified with **Ctrl**, put unprintable characters in the `Character` field.
- **Ctrl**, **Alt**, **Shift**, function and several other keys generate no `Character` field data.

Modifier Keys

When you press and release a key and a modifier key, the modifier key is returned in the event structure `Modifier` field. If you press and release a modifier key only, its name is not returned in the event structure of the `KeyReleaseFcn`, but is returned in the event structure of the `KeyPressFcn`.

Explore KeyReleaseFcn Behavior

The following code example creates a figure and defines a function handle callback for the `KeyReleaseFcn` property which reports the event data that the callback receives.

- Click here to view the example in the MATLAB editor
- Click here to add the example to the MATLAB path
- Click to run the example — Press and release various key combinations while the figure has focus. The callback displays event data in the Command Window.

```
function key_releaseFcn
   figure('KeyReleaseFcn',@cb)
      function cb(src,evnt)
      if ~isempty(evnt.Modifier)
         for ii = 1:length(evnt.Modifier)
            out = sprintf('Character: %c\nModifier: %s\nKey: %s\n',...
     evnt.Character,evnt.Modifier{ii},evnt.Key);
            disp(out)
         end
      else
         out = sprintf('Character: %c\nModifier: %s\nKey: %s\n',...
    evnt.Character,'No modifier key',evnt.Key);
         disp(out)
      end
   end
end
```

`MenuBar`

`none | {figure}`

Enable-disable figure menu bar. Enables you to display or hide the menu bar that MATLAB places at the top of a figure window. The default (`figure`) is to display the menu bar.

This property affects only built-in menus. This property does not affect menus defined with the `uimenu` command.

Changing the `WindowStyle` of a window to `'modal'` hides both its toolbar and menu bar, if they exist. Changing `WindowStyle` from `'modal'` to `'normal'` or `'docked'` displays any toolbar or menu bar a figure has.

`MinColormap`

scalar (default = `64`)

Minimum number of color table entries used. Specifies the minimum number of system color table entries used by MATLAB to store the colormap defined for the figure (see the `Colormap`

property). In certain situations, you might need to increase this value to ensure proper use of colors.

For example, suppose you run color-intensive applications in addition to MATLAB and have defined a large figure colormap (for example, 150 to 200 colors). MATLAB might select colors that are close but not exact from the existing colors in the system color table because there are not enough slots available to define all the colors you specified.

To ensure that MATLAB uses exactly the colors you define in the figure colormap, set `MinColormap` equal to the length of the colormap.

```
set(gcf,'MinColormap',length(get(gcf,'ColorMap')))
```

Note that the larger the value of `MinColormap`, the greater the likelihood that other windows (including other MATLAB figure windows) will be displayed in false colors.

Note The `MinColormap` property is being deprecated and will be removed in a future release

---

`Name`
: `string`

Figure window title. Specifies the title displayed in the figure window. By default, `Name` is empty and the figure title is displayed as `Figure 1`, `Figure 2`, and so on. When you set this parameter to a string, the figure title becomes `Figure 1: <string>`. See the `NumberTitle` property.

`NextPlot`
: `new | {add} | replace | replacechildren`

How to add next plot. Determines which figure MATLAB uses to display graphics output. If the value of the current figure is:

- `new` — Create a new figure to display graphics (unless an existing parent is specified in the graphing function as a property/value pair).
- `add` — Use the current figure to display graphics (the default).
- `replace` — Reset all figure properties except `Position` to their defaults and delete all figure children before displaying graphics (equivalent to `clf reset`).
- `replacechildren` — Remove all child objects, but do not reset figure properties (equivalent to `clf`).

The `newplot` function provides an easy way to handle the `NextPlot` property. For more information, see the axes `NextPlot` property and "Controlling Graphics Output".

`NumberTitle`
`{on} | off` (GUIDE default `off`)

Figure window title number. Determines whether the string `Figure No.  N` (where `N` is the figure number) is prefixed to the figure window title. See the `Name` property.

`OuterPosition`
four-element vector

Figure position including title bar, menu bar, tool bars, and outer edges. Specifies the size and location on the screen of the full figure window including the title bar, menu bar, tool bars, and outer edges. Specify the position rectangle with a four-element vector of the form:

```
rect = [left, bottom, width, height]
```

where `left` and `bottom` define the distance from the lower-left corner of the screen to the lower-left corner of the full figure

window. `width` and `height` define the dimensions of the window. See the `Units` property for information on the units used in this specification. The `left` and `bottom` elements can be negative on systems that have more than one monitor.

Position of Docked Figures

If the figure is docked in the MATLAB desktop, then the `OuterPosition` property is specified with respect to the figure group container instead of the screen.

Moving and Resizing Figures

Use the `get` function to obtain this property and determine the position of the figure. Use the `set` function to resize and move the figure to a new location. You cannot set the figure `OuterPosition` when it is docked.

Note On Microsoft Windows systems, figure windows cannot be less than `104` pixels wide, regardless of the value of the `OuterPosition` property.

`PaperOrientation`
`{portrait} | landscape`

Horizontal or vertical paper orientation. Determines how to orient printed figures on the page.

- `portrait` — Orients the longest page dimension vertically.
- `landscape` — Orients the longest page dimension horizontally.

See the `orient` command for more information.

`PaperPosition`
four-element rect vector

Location on printed page. A rectangle that determines the location of the figure on the printed page. Specify this rectangle with a vector of the form:

```
rect = [left, bottom, width, height]
```

where `left` specifies the distance from the left side of the paper to the left side of the rectangle and `bottom` specifies the distance from the bottom of the page to the bottom of the rectangle. Together these distances define the lower-left corner of the rectangle. `width` and `height` define the dimensions of the rectangle. The `PaperUnits` property specifies the units used to define this rectangle.

`PaperPositionMode`
`auto | {manual}`

WYSIWYG printing of figure.

- `manual` — MATLAB honors the value specified by the `PaperPosition` property.
- `auto` — MATLAB prints the figure the same size as it appears on the computer screen, centered on the page.

`PaperSize`
`[width height]`

Paper size. Size of the current `PaperType`, measured in `PaperUnits`. See `PaperType` to select standard paper sizes.

`PaperType`
Select a value from the following table.

Selection of standard paper size. Sets the `PaperSize` to one of the following standard sizes.

Paper Sizes Table

Property Value	Size (Width x Height)
usletter (default)	8.5-by-11 inches
uslegal	8.5-by-14 inches
tabloid	11-by-17 inches
A0	841-by-1189 mm
A1	594-by-841 mm
A2	420-by-594 mm
A3	297-by-420 mm
A4	210-by-297 mm
A5	148-by-210 mm
B0	1029-by-1456 mm
B1	728-by-1028 mm
B2	514-by-728 mm
B3	364-by-514 mm
B4	257-by-364 mm
B5	182-by-257 mm
arch-A	9-by-12 inches
arch-B	12-by-18 inches
arch-C	18-by-24 inches
arch-D	24-by-36 inches
arch-E	36-by-48 inches
A	8.5-by-11 inches
B	11-by-17 inches
C	17-by-22 inches

Property Value	Size (Width x Height)
`D`	22-by-34 inches
`E`	34-by-43 inches

Note that you might need to change the `PaperPosition` property in order to position the printed figure on the new paper size. One solution is to use `normalized` `PaperUnits`, which enables MATLAB to automatically size the figure to occupy the same relative amount of the printed page, regardless of the paper size.

`PaperUnits`
`normalized | {inches} | centimeters | points`

Hardcopy measurement units. Specifies the units used to define the `PaperPosition` and `PaperSize` properties. MATLAB measures all units from the lower-left corner of the page. `normalized` units map the lower-left corner of the page to (0, 0) and the upper-right corner to (1.0, 1.0). `inches`, `centimeters`, and `points` are absolute units (one point equals 1/72 of an inch).

If you change the value of `PaperUnits`, it is good practice to return the property to its default value after completing your computation so as not to affect other functions that assume `PaperUnits` is set to the default value.

`Parent`
`handle`

Handle of figure's parent. The parent of a figure object is the `root object` object. The handle to the root is always 0.

`Pointer`

```
crosshair | {arrow} | watch | topl |
topr | botl | botr | circle | cross |
fleur | left | right | top | bottom |
fullcrosshair | ibeam | custom | hand
```

Pointer symbol selection. Determines the symbol used to indicate the pointer (cursor) position in the figure window. Setting `Pointer` to `custom` allows you to define your own pointer symbol. See the `PointerShapeCData` property and "Specifying the Figure Pointer" for more information.

`PointerShapeCData`
16-by-16 matrix

User-defined pointer. Defines the pointer used when you set the `Pointer` property to `custom`. It is a 16-by-16 element matrix defining the 16-by-16 pixel pointer using the following values:

- `1` — Color pixel black.
- `2` — Color pixel white.
- `NaN` — Make pixel transparent (underlying screen shows through).

Element `(1,1)` of the `PointerShapeCData` matrix corresponds to the upper-left corner of the pointer. Setting the `Pointer` property to one of the predefined pointer symbols does not change the value of the `PointerShapeCData`. Computer systems supporting 32-by-32 pixel pointers fill only one quarter of the available pixmap.

`PointerShapeHotSpot`
two-element vector

Pointer active area. Specifies the row and column indices in the `PointerShapeCData` matrix defining the pixel indicating the pointer location. The location is contained in the `CurrentPoint` property and the root object's `PointerLocation` property. The default value is element `(1,1)`, which is the upper-left corner.

`Position`
four-element vector

Figure position. Specifies the size and location on the screen of the figure window, not including title bar, menu bar, tool bars, and outer edges. Specify the position rectangle with a four-element vector of the form:

```
rect = [left, bottom, width, height]
```

where `left` and `bottom` define the distance from the lower-left corner of the screen to the lower-left corner of the figure window. `width` and `height` define the dimensions of the window. See the `Units` property for information on the units used in this specification. The `left` and `bottom` elements can be negative on systems that have more than one monitor.

Position of Docked Figures

If the figure is docked in the MATLAB desktop, then the `Position` property is specified with respect to the figure group container instead of the screen.

Moving and Resizing Figures

You can use the `get` function to obtain this property and determine the position of the figure and you can use the `set` function to resize and move the figure to a new location. You cannot set the figure `Position` when it is docked.

Note On Windows systems, figure windows cannot be less than 104 pixels wide, regardless of the value of the `Position` property.

Also, the figure window includes the area to which MATLAB can draw; it does not include the title bar, menu bar, tool bars, and outer edges. To place the full window, use the `OuterPosition` property.

`Renderer`

`painters | zbuffer | OpenGL`

Rendering method used for screen and printing. Selects the method used to render MATLAB graphics. The choices are:

- `painters` — The original rendering method used by MATLAB is faster when the figure contains only simple or small graphics objects.
- `zbuffer` — MATLAB draws graphics objects faster and more accurately because it colors objects on a per-pixel basis and MATLAB renders only those pixels that are visible in the scene (thus eliminating front-to-back sorting errors). Note that this method can consume a lot of system memory if MATLAB is displaying a complex scene.
- `OpenGL` — OpenGL is a renderer that is available on many computer systems. This renderer is generally faster than `painters` or `zbuffer` and in some cases enables MATLAB to access graphics hardware that is available on some systems.

Hardware vs. Software OpenGL Implementations

There are two kinds of OpenGL implementations — hardware and software.

The hardware implementation uses special graphics hardware to increase performance and is therefore significantly faster than the software version. Many computers have this special hardware available as an option or might come with this hardware right out of the box.

Software implementations of OpenGL are much like the ZBuffer renderer that is available on MATLAB Version 5.0 and later; however, OpenGL generally provides superior performance to ZBuffer.

OpenGL Availability

OpenGL is available on all computers that run MATLAB. MATLAB automatically finds hardware-accelerated versions of OpenGL if such versions are available. If the hardware-accelerated version is not available, then MATLAB uses the software version (except on Macintosh systems, which do not support software OpenGL).

The following software versions are available:

- On UNIX systems, MATLAB uses the software version of OpenGL that is included in the MATLAB distribution.
- On Windows, OpenGL is available as part of the operating system. If you experience problems with OpenGL, contact your graphics driver vendor to obtain the latest qualified version of OpenGL.
- On Macintosh systems, software OpenGL is not available.

MATLAB issues a warning if it cannot find a usable OpenGL library.

Selecting Hardware-Accelerated or Software OpenGL

MATLAB enables you to switch between hardware-accelerated and software OpenGL. However, Windows and UNIX systems behave differently:

- On Windows systems, you can toggle between software and hardware versions any time during the MATLAB session.
- On UNIX systems, you must set the OpenGL version before MATLAB initializes OpenGL. Therefore, you cannot issue the `opengl info` command or create graphs before you call `opengl software`. To reenable hardware accelerated OpenGL, you must restart MATLAB.
- On Macintosh systems, software OpenGL is not available.

If you do not want to use hardware OpenGL, but do want to use object transparency, you can issue the following command.

```
opengl software
```

This command forces MATLAB to use software OpenGL. Software OpenGL is useful if your hardware-accelerated version of OpenGL does not function correctly and you want to use image, patch, or surface transparency, which requires the OpenGL renderer. To reenable hardware OpenGL, use the command:

```
opengl hardware
```

on Windows systems or restart MATLAB on UNIX systems.

By default, MATLAB uses hardware-accelerated OpenGL.

See the `opengl` reference page for additional information

Determining the OpenGL Library Version

To determine the version and vendor of the OpenGL library that MATLAB is using on your system, type the following command at the MATLAB prompt:

```
opengl info
```

The returned information contains a line that indicates if MATLAB is using software (`Software = true`) or hardware-accelerated (`Software = false`) OpenGL.

This command also returns a string of extensions to the OpenGL specification that are available with the particular library MATLAB is using. Include this information if you report a bug.

Note that issuing the `opengl info` command causes MATLAB to initialize OpenGL.

OpenGL vs. Other MATLAB Renderers

There are some differences between drawings created with OpenGL and those created with other renderers. The OpenGL specific differences include:

- OpenGL does not do colormap interpolation. If you create a surface or patch using indexed color and interpolated face or edge coloring, OpenGL interpolates the colors through the RGB color cube instead of through the colormap.
- OpenGL does not support the `phong` value for the `FaceLighting` and `EdgeLighting` properties of surfaces and patches.
- OpenGL does not support logarithmic-scale axes.
- OpenGL and Zbuffer renderers display objects sorted in front to back order, as seen on the monitor, and lines always draw in front of faces when at the same location on the plane of the monitor. Painters sorts by child order (order specified).

XServer Connection Lost

When using Linux or Macintosh systems, MATLAB can crash with a segmentation violation if the connection to the XServer is broken. If this occurs, ensure that the system has the latest XServer installed.

You can also try using software OpenGL and upgrade the OpenGL driver on a Linux system.

Try these workarounds:

1. Upgrade you XServer to the latest version
2. Upgrade your OpenGL driver to the latest version
3. Switch to software OpenGL by entering this command:

   ```
   opengl software
   ```

`RendererMode`
`{auto} | manual`

Automatic or user selection of renderer. Specifies whether MATLAB should choose the `Renderer` based on the contents of the figure window, or whether the `Renderer` should remain unchanged.

When the `RendererMode` property is `auto`, MATLAB selects the rendering method for printing as well as for screen display based on the size and complexity of the graphics objects in the figure.

For printing, MATLAB switches to `zbuffer` at a greater scene complexity than for screen rendering because printing from a z-buffered figure can be considerably slower than one using the `painters` rendering method, and can result in large PostScript® files. However, the output does always match what is on the screen. The same holds true for OpenGL: the output is the same as that produced by the `zbuffer` renderer — a bitmap with a resolution determined by the `print` command's `-r` option.

Criteria for Autoselection of the OpenGL Renderer

When the `RendererMode` property is `auto`, MATLAB uses the following criteria to determine whether to select the OpenGL renderer:

If the `opengl` autoselection mode is `autoselect`, MATLAB selects OpenGL if

- The host computer has OpenGL installed and is in True Color mode (OpenGL does not fully support 8-bit color mode).
- The figure contains no logarithmic axes (OpenGL does not support logarithmic axes).
- MATLAB would select `zbuffer` based on figure contents.

- Patch objects' faces have no more than three vertices (some OpenGL implementations of patch tessellation are unstable).
- The figure contains less than 10 uicontrols (OpenGL clipping around uicontrols is slow).
- No line objects use markers (drawing markers is slow).
- You do not specify Phong lighting (OpenGL does not support Phong lighting; if you specify Phong lighting, MATLAB uses the ZBuffer renderer).

  Or

- Figure objects use transparency (OpenGL is the only MATLAB renderer that supports transparency).

When the `RendererMode` property is `manual`, MATLAB does not change the `Renderer`, regardless of changes to the figure contents.

`Resize`
`{on} | off`

Window resize mode. Determines if you can resize the figure window with the mouse. `on` means you can resize the window, `off` means you cannot. When `Resize` is `off`, the figure window does not display any resizing controls (such as boxes at the corners), to indicate that it cannot be resized.

`ResizeFcn`
function handle | cell array containing function handle and additional arguments | string (not recommended)

Window resize callback function. Executes whenever you resize the figure window and also when the figure is created. You can query the figure's `Position` property to determine the new size and position of the figure. During execution of the callback routine, the handle to the figure being resized is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

You can use ResizeFcn to maintain a GUI layout that is not directly supported by the MATLAB Position/Units paradigm.

For example, consider a GUI layout that maintains an object at a constant height in pixels and attached to the top of the figure, but always matches the width of the figure. The following ResizeFcn accomplishes this; it keeps the uicontrol whose Tag is 'StatusBar' 20 pixels high, as wide as the figure, and attached to the top of the figure. Note the use of the Tag property to retrieve the uicontrol handle, and the gcbo function to retrieve the figure handle. Also note the defensive programming regarding figure Units, which the callback requires to be in pixels in order to work correctly, but which the callback also restores to their previous value afterwards.

```
u = findobj('Tag','StatusBar');
fig = gcbo;
old_units = get(fig,'Units');
set(fig,'Units','pixels');
figpos = get(fig,'Position');
upos = [0, figpos(4) - 20, figpos(3), 20];
set(u,'Position',upos);
set(fig,'Units',old_units);
```

You can change the figure Position from within the ResizeFcn callback; however, the ResizeFcn is not called again as a result.

Note that the print command can cause the ResizeFcn to be called if the PaperPositionMode property is manual and you have defined a resize function. If you do not want your resize function called by print, set the PaperPositionMode to auto.

See “Introduction” for an example of how to implement a resize function for a GUI.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`Selected`
`on | off`

Is object selected? This property indicates whether the figure is selected. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`
`{on} | off`

Figures do not indicate selection.

`SelectionType`
`{normal} | extend | alt | open`

Mouse selection type. MATLAB maintains this property to provide information about the last mouse button press that occurred within the figure window. This information indicates the type of selection made. Selection types are actions that MATLAB generally associates with particular responses from the user interface software (for example, single-clicking a graphics object places it in move or resize mode; double-clicking a file name opens it, etc.).

The physical action required to make these selections varies on different platforms. However, all selection types exist on all platforms.

Selection Type	Microsoft Windows	X-Windows
`Normal`	Click left mouse button.	Click left mouse button.
`Extend`	**Shift** - click left mouse button or click both left and right mouse buttons.	**Shift**-click left mouse button or click middle mouse button.

Selection Type	Microsoft Windows	X-Windows
`Alternate`	**Control** - click left mouse button or click right mouse button.	**Control**-click left mouse button or click right mouse button.
`Open`	Double-click any mouse button.	Double-click any mouse button.

Note For `uicontrols` whose `Enable` property is `on`, a single left-click, **Ctrl**-left click, or **Shift**-left click sets the figure `SelectionType` property to `normal`. For a list box `uicontrol` whose `Enable` property is `on`, the second click of a double-click sets the figure `SelectionType` property to `open`. All clicks on `uicontrols` whose `Enable` property is `inactive` or `off` and all right-clicks on `uicontrols` whose `Enable` property is `on` set the figure `SelectionType` property as specified in the preceding table.

`Tag`

string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

For example, suppose you want to direct all graphics output from a file to a particular figure, regardless of user actions that might have changed the current figure. To do this, identify the figure with a `Tag`.

```
figure('Tag','Plotting Figure')
```

Then make that figure the current figure before drawing by searching for the `Tag` with `findobj`.

```
figure(findobj('Tag','Plotting Figure'))
```

`Toolbar`
`none | {auto} | figure`

Control display of figure toolbar. Control whether MATLAB displays the default figure toolbar on figures. The possible values are:

- `none` — Do not display the figure toolbar.
- `auto` — Display the figure toolbar, but remove it if a uicontrol is added to the figure.
- `figure` — Display the figure toolbar.

Note that this property affects only the figure toolbar; it does not affect other toolbars (for example, the Camera Toolbar or Plot Edit Toolbar). Selecting **Figure Toolbar** from the figure **View** menu sets this property to `figure`.

Changing the `WindowStyle` of a window to `'modal'` hides both its toolbar and menu bar, if they exist. Changing `WindowStyle` from `'modal'` to `'normal'` or `'docked'` displays any toolbar or menu bar a figure has.

`Type`
string (read-only)

Object class. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For figures, `Type` is always '`figure`'.

`UIContextMenu`
handle of uicontextmenu object

Associate a context menu with the figure. Assign this property the handle of a uicontextmenu object created in the figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the figure.

`Units`

`inches | centimeters | normalized | points | {pixels} | characters`

Units of measurement. Specifies the units MATLAB uses to interpret size and location data. All units are measured from the lower-left corner of the window.

- `normalized` — Units map the lower-left corner of the figure window to (0,0) and the upper-right corner to (1.0,1.0).
- `inches`, `centimeters`, and `points` — Absolute units. 1 point = $^1/_{72}$ inch.
- `pixels` — Size depends on screen resolution.
- `characters` — Based on the size of characters in the default system font. The width of one `characters` unit is the width of the letter `x`, and the height of one `characters` unit is the distance between the baselines of two lines of text.

This property affects the `CurrentPoint` and `Position` properties. If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is the default value.

When specifying the units as property/value pairs during object creation, you must set the `Units` property before specifying the properties that you want to use these units.

`UserData`

matrix

User-specified data. Data you want to associate with the figure object. The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`
`{on} | off`

Object visibility. The `Visible` property determines whether an object is displayed on the screen. If the `Visible` property of a figure is `off`, the entire figure window is invisible.

A Note About Using the Window Button Properties

Your window button callback functions might need to update the display by calling `drawnow` or `pause`, which causes MATLAB to process all events in the queue. Processing the event queue can cause your window button callback functions to be reentered. For example, a `drawnow` in the `WindowButtonDownFcn` might result in the `WindowButtonDownFcn` being called again before the first call has finished. You should design your code to handle reentrancy and you should not depend on global variables that might change state during reentrance.

You can use the `Interruptible` and `BusyAction` figure properties to control how events interact.

`WindowButtonDownFcn`
function handle | cell array containing function handle and additional arguments | string (not recommended)

Button press callback function. Executes whenever you press a mouse button while the pointer is in the figure window. See the `WindowButtonMotionFcn` property for an example.

Note When using a two- or three-button mouse on Macintosh systems, right-button and middle-button presses are not always reported. This happens *only* when a new figure window appears under the mouse cursor and the mouse is clicked without first moving it. In this circumstance, for the `WindowButtonDownFcn` to work, the user needs to do one of the following:

- Move the mouse after the figure is created, then click any mouse button
- Press **Shift** or **Ctrl** while clicking the left mouse button to perform the Extend and Alternate selection types

Pressing the left mouse button (or single mouse button) works without having to take either of the above actions.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`WindowButtonMotionFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Mouse motion callback function. Executes whenever you move the pointer within the figure window. Define the `WindowButtonMotionFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with key release and an event structure).

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

Example Using All Window Button Properties

Click to view in editor — This example enables you to use mouse motion to draw lines. It uses all three window button functions.

Click to run example — Click the left mouse button in the axes and move the cursor, left-click to define the line end point, right-click to end drawing mode.

Note On some computer systems, the `WindowButtonMotionFcn` is executed when a figure is created even though there has been no mouse motion within the figure.

`WindowButtonUpFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Button release callback function. Executes whenever you release a mouse button. Define the `WindowButtonUpFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with key release and an event structure).

The button up event is associated with the figure window in which the preceding button down event occurred. Therefore, the pointer need not be in the figure window when you release the button to generate the button up event.

If the callback routines defined by `WindowButtonDownFcn` or `WindowButtonMotionFcn` contain `drawnow` commands or call other functions that contain `drawnow` commands and the `Interruptible` property is `off`, the `WindowButtonUpFcn` might not be called. You can prevent this problem by setting `Interruptible` to `on`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`WindowKeyPressFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Key press callback function for the figure window. Executes whenever a key press occurs. This is a callback function invoked by a key press that occurs while either the figure window or any of its children has focus. Define the `WindowKeyPressFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with key release and an event structure).

See Function Handle Callbacks for information on how to use function handles to define the callback function.

When no callback is specified for this property (which is the default state), MATLAB passes any key presses to the Command Window. However, when you define a callback for this property, the figure retains focus with each key press and executes the specified callback with each key press.

WindowKeyPressFcn Event Structure

When you specify the callback as a function handle, MATLAB passes to it a structure to containing the following fields.

Field	Contents
`Character`	The character displayed as a result of the releasing the key(s), which can be empty or unprintable
`Key`	The key being released, identified by the lowercase label on key or a descriptive string
`Modifier`	A cell array containing the names of one or more modifier keys being released (i.e., **control**, **alt**, **shift**). On Macintosh computers, it contains `'command'` when releasing the **command** modifier key

For more information and examples of use, see the `KeyPressFcn` property description.

`WindowKeyReleaseFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Key release callback function for the figure window. Executes whenever a key release occurs. This is a callback function invoked by a key release that occurs while the figure window or any of its children has focus. Define the `WindowKeyReleaseFcn` as a function handle. The function must define at least two input arguments (handle of the figure associated with key release and an event structure).

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

WindowKeyReleaseFcn Event Structure

When you specify the callback as a function handle, MATLAB passes to it a structure to containing the following fields.

Field	Contents
`Character`	The character displayed as a result of the releasing the key(s), which can be empty or unprintable
`Key`	The key being released, identified by the lowercase label on key or a descriptive string
`Modifier`	A cell array containing the names of one or more modifier keys being released (i.e., **control**, **alt**, **shift**). On Macintosh computers, it contains `'command'` when releasing the **command** modifier key

For more information and examples of use, see the `KeyReleaseFcn` property description.

`WindowScrollWheelFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Respond to mouse scroll wheel. Executes when the mouse wheel is scrolled while the figure has focus. MATLAB executes the callback with each single mouse wheel click.

Note that it is possible for another object to capture the event from MATLAB. For example, if the figure contains Java or ActiveX control objects that are listening for mouse scroll wheel events, then these objects can consume the events and prevent the `WindowScrollWheelFcn` from executing.

There is no default callback defined for this property.

WindowScrollWheelFcn Event Structure

When the callback is a function handle, MATLAB passes a structure to the callback function that contains the following fields.

Field	Contents
`VerticalScrollCount`	A positive or negative integer that indicates the number of scroll wheel clicks. Positive values indicate clicks of the wheel scrolled in the down direction. Negative values indicate clicks of the wheel scrolled in the up direction.
`VerticalScrollAmount`	The current system setting for the number of lines that are scrolled for each click of the scroll wheel. If the mouse property setting for scrolling is set to `One screen at a time`, `VerticalScrollAmount` returns a value of 1.

Effects on Other Properties

- `CurrentObject` property — Mouse scrolling does not update this figure property.
- `CurrentPoint` property — If there is no callback defined for the `WindowScrollWheelFcn` property, then MATLAB does not update the `CurrentPoint` property as the scroll wheel is turned. However, if there is a callback defined for the `WindowScrollWheelFcn` property, then MATLAB updates the `CurrentPoint` property just before executing the callback. This enables you to determine the point at which the mouse scrolling occurred.
- `HitTest` property — The `WindowScrollWheelFcn` callback executes regardless of the setting of the figure `HitTest` property.
- `SelectionType` property — The `WindowScrollWheelFcn` callback has no effect on this property.

Values Returned by VerticalScrollCount

When a user moves the mouse scroll wheel by one click, MATLAB increments the count by +/- 1, depending on the direction of the scroll (scroll down being positive). When MATLAB calls the `WindowScrollWheelFcn` callback, the counter is reset. In most cases, this means that the absolute value of the returned value is 1. However, if the `WindowScrollWheelFcn` callback takes a long enough time to return and/or the user spins the scroll wheel very fast, then the returned value can have an absolute value greater than one.

The actual value returned by `VerticalScrollCount` is the algebraic sum of all scroll wheel clicks that occurred since last processed. This enables your callback to respond correctly to the user's action.

Example

Click to view in editor — This example creates a graph of a function and enables you to use the mouse scroll wheel to change the range over which a mathematical function is evaluated and update the graph to reflect the new limits as you turn the scroll wheel.

Click to run example — Mouse over the figure and scroll your mouse wheel.

Related Information

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`WindowStyle`

`{normal} | modal | docked`

Normal, modal, or dockable window behavior. When `WindowStyle` is `modal`:

- The figure window traps all keyboard and mouse events over all MATLAB windows as long as they are visible.
- Windows belonging to applications other than MATLAB are unaffected.
- Modal figures remain stacked above all normal figures and the MATLAB Command Window.
- When multiple modal windows exist, the most recently created window keeps focus and stays above all other windows until it becomes invisible, or is returned to `WindowStyle normal`, or is deleted. At that time, focus reverts to the window that last had focus.

Use modal figures to create dialog boxes that force the user to respond without being able to interact with other windows. Typing **Ctrl+C** while the figure has focus causes all figures with

`WindowStyle modal` to revert to `WindowStyle normal`, allowing you to type at the command line.

Invisible Modal Figures

Figures with `WindowStyle modal` and `Visible` do not behave modally until they are made visible, so it is acceptable to hide a `modal` window for later reuse instead of destroying it.

Stacking Order of Modal Figures

Creating a figure with `WindowStyle modal` stacks it on top of all existing figure windows, making them inaccessible as long as the top figure exists and remains modal. However, any new figures created after a modal figure is displayed (for example, plots created by a modal GUI) stack on top of it and are accessible; they can be modal as well.

Changing Modes

You can change the `WindowStyle` of a figure at any time, including when the figure is visible and contains children. However, on some systems this might cause the figure to flash or disappear and reappear, depending on the windowing system's implementation of normal and modal windows. For best visual results, you should set `WindowStyle` at creation time or when the figure is invisible.

Window Decorations on Modal Figures

Modal figures do not display `uimenu` children, built-in menus, or toolbars but it is not an error to create uimenus in a modal figure or to change `WindowStyle` to `modal` on a figure with uimenu children. The uimenu objects exist and their handles are retained by the figure. If you reset the figure's `WindowStyle` to `normal`, the uimenus are displayed.

Docked WindowStyle

When `WindowStyle` is `docked`, the figure is docked in the desktop or a document window. When you issue the following command:

```
set(figure_handle,'WindowStyle','docked')
```

MATLAB docks the figure identified by *`figure_handle`* and sets the `DockControls` property to `on`, if it was `off`.

Note that if `WindowStyle` is `docked`, you cannot set the `DockControls` property to `off`.

The value of the `WindowStyle` property is not changed by calling `reset` on a figure.

`WVisual`
identifier string (Windows only)

Specify pixel format for figure. MATLAB automatically selects a pixel format for figures based on your current display settings, the graphics hardware available on your system, and the graphical content of the figure.

Usually, MATLAB chooses the best pixel format to use in any given situation. However, in cases where graphics objects are not rendered correctly, you might be able to select a different pixel format and improve results. See "Understanding the WVisual String" for more information.

Querying Available Pixel Formats on Window Systems

You can determine what pixel formats are available on your system for use with MATLAB using the following statement:

```
set(gcf,'WVisual')
```

MATLAB returns a list of the currently available pixel formats for the current figure. For example, the following are the first three entries from a typical list:

01 (RGB 16 bits(05 06 05 00) zdepth 24, Hardware Accelerated, OpenGL, GDI, Window)

02 (RGB 16 bits(05 06 05 00) zdepth 24, Hardware Accelerated, OpenGL, Double Buffered, Window)

03 (RGB 16 bits(05 06 05 00) zdepth 24, Hardware Accelerated, OpenGL, Double Buffered, Window)

Use the number at the beginning of the string to specify which pixel format to use. For example:

```
set(gcf,'WVisual','02')
```

specifies the second pixel format in the list above. Note that pixel formats might differ on your system.

Understanding the WVisual String

The string returned by querying the `WVisual` property provides information on the pixel format. For example:

- `RGB 16 bits(05 06 05 00)` — Indicates true color with 16-bit resolution (5 bits for red, 6 bits for green, 5 bits for blue, and 0 for alpha (transparency). MATLAB requires true color.
- `zdepth 24` — Indicates 24-bit resolution for sorting object's front to back position on the screen. Selecting pixel formats with higher (24 or 32) zdepth might solve sorting problems.
- `Hardware Accelerated` — Some graphics functions might be performed by hardware for increased speed. If there are incompatibilities between your particular graphic hardware and MATLAB, select a pixel format in which the term `Generic` appears instead of `Hardware Accelerated`.
- `Opengl` — Supports OpenGL. See the preceding "Pixel Formats and OpenGL" for more information.
- `GDI` — Supports for Windows 2-D graphics interface.

- `Double Buffered` — Support for double buffering with the OpenGL renderer. Note that the figure `DoubleBuffer` property applies only to the painters renderer.
- `Bitmap` — Support for rendering into a bitmap (as opposed to drawing in the window).
- `Window` — Support for rendering into a window.

Pixel Formats and OpenGL

If you are experiencing problems using hardware OpenGL on your system, you can try using generic OpenGL, which is implemented in software. To do this, first instruct MATLAB to use the software version of OpenGL with the following statement:

```
opengl software
```

Then allow MATLAB to select best pixel format to use.

See the `Renderer` property for more information on how MATLAB uses OpenGL.

`WVisualMode`
`{auto} | manual` (Windows only)

Auto or manual selection of pixel format. `WVisualMode` can take on two values — `auto` (the default) and `manual`. In `auto` mode, MATLAB selects the best pixel format to use based on your computer system and the graphical content of the figure. In `manual` mode, MATLAB does not change the visual from the one currently in use. Setting the `WVisual` property sets this property to `manual`.

`XDisplay`
display identifier (UNIX only)

Contains the display used for MATLAB. You can query this property to determine the name of the display that MATLAB is

using. For example, if MATLAB is running on a system called `mycomputer`, querying `XDisplay` returns a string of the following form:

```
get(gcf,'XDisplay')
ans
mycomputer:0.0
```

Setting XDisplay on Motif

If your computer uses Motif-based figures, you can specify the display MATLAB uses for a figure by setting the value of the figure's `XDisplay` property. For example, to display the current figure on a system called `fred`, use the command:

```
set(gcf,'XDisplay','fred:0.0')
```

`XVisual`
visual identifier (UNIX only)

Select visual used by MATLAB. You can select the visual used by MATLAB by setting the `XVisual` property to the desired visual ID. This can be useful if you want to test your application on an 8-bit or grayscale visual. To see what visuals are available on your system, use the UNIX `xdpyinfo` command. From MATLAB, type:

```
!xdpyinfo
```

The information returned contains a line specifying the visual ID. For example:

```
visual id:    0x23
```

To use this visual with the current figure, set the `XVisual` property to the ID.

```
set(gcf,'XVisual','0x23')
```

To see which of the available visuals MATLAB can use, call `set` on the `XVisual` property:

```
set(gcf,'XVisual')
```

The following typical output shows the visual being used (in curly braces) and other possible visuals. Note that MATLAB requires a TrueColor visual.

```
{ 0x23 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff) }
  0x24 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
  0x25 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
  0x26 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
  0x27 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
  0x28 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
  0x29 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
  0x2a (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
```

You can also use the `glxinfo` UNIX command to see what visuals are available for use with the OpenGL renderer. From MATLAB, type:

```
!glxinfo
```

After providing information about the implementation of OpenGL on your system, `glxinfo` returns a table of visuals. The partial listing below shows typical output:

```
visual  x  bf lv rg d st colorbuffer ax dp st accumbuffer  ms  cav
 id dep cl sp sz l  ci b ro  r  g  b  a bf th cl  r  g  b  a ns b eat
----------------------------------------------------------------
-
0x23 24 tc  0 24  0 r  y  .  8  8  8  8  0  0  0  0  0  0  0  0 0 None
0x24 24 tc  0 24  0 r  .  .  8  8  8  8  0  0  0  0  0  0  0  0 0 None
0x25 24 tc  0 24  0 r  y  .  8  8  8  8  0 24  8  0  0  0  0  0 0 None
0x26 24 tc  0 24  0 r  .  .  8  8  8  8  0 24  8  0  0  0  0  0 0 None
0x27 24 tc  0 24  0 r  y  .  8  8  8  8  0  0  0 16 16 16  0  0 0 Slow
```

The third column is the class of visual. `tc` means a true color visual. Note that some visuals might be labeled `Slow` under the caveat column. Such visuals should be avoided.

To determine which visual MATLAB will use by default with the OpenGL renderer, use the MATLAB `opengl info` command. The returned entry for the visual might look like the following:

```
Visual = 0x23 (TrueColor, depth 24, RGB mask 0xff0000 0xff00 0x00ff)
```

Experimenting with a different TrueColor visual might improve certain rendering problems.

`XVisualMode`

`{auto} | manual` (UNIX only)

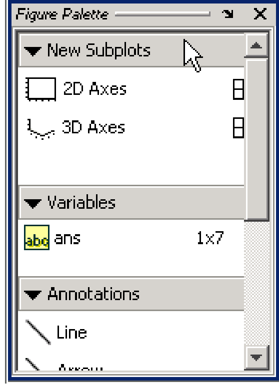
Auto or manual selection of visual. `XVisualMode` can take on two values — `auto` (the default) and `manual`. In `auto` mode, MATLAB selects the best visual to use based on the number of colors, availability of the OpenGL extension, etc. In `manual` mode, MATLAB does not change the visual from the one currently in use. Setting the `XVisual` property sets this property to `manual`.

See Also

`figure`

Purpose

Show or hide **Figure Palette**



Syntax

```
figurepalette('show')
figurepalette('hide')
figurepalette
figurepalette(figure_handle,...)
```

Description

`figurepalette('show')` displays the palette on the current figure.

`figurepalette('hide')` hides the palette on the current figure.

`figurepalette` toggles the visibility of the palette on the current figure. You can also use `figurepalette('toggle')` instead for the same functionality.

`figurepalette(figure_handle,...)` shows or hides the palette on the figure specified by `figure_handle`.

Tips

If you call `figurepalette` in a MATLAB program and subsequent lines depend on the Figure Palette being fully initialized, follow it by `drawnow` to ensure complete initialization.

Alternatives

To collectively enable **Plotting Tools**, use the large Plotting Tool icon on the figure toolbar. To collectively disable the **Plotting Tools**,

use the smaller icon . Open or close the **Figure Palette** tool from the figure's **View** menu.

See Also

`plottools` | `plotbrowser` | `propertyeditor`

Purpose

Set or get attributes of file or folder

Syntax

```
fileattrib
fileattrib(name)

fileattrib(name,attribs)
fileattrib(name,attribs,users)
fileattrib(name,attribs,users,'s')

[status,message,messageid] = fileattrib(name,attribs,___)
[status,message] = fileattrib(name)
```

Description

`fileattrib` gets attribute values for the current folder, using the following structure, where `Name` is always a string containing the current folder name. For the other fields, a value of `0` indicates that the attribute is off, `1` indicates that the attribute is on, and `NaN` indicates that the attribute does not apply:

```
Name
archive
system
hidden
directory
UserRead
UserWrite
UserExecute
GroupRead
GroupWrite
GroupExecute
OtherRead
```

`fileattrib(name)` gets the attribute values for the named file or folder.

`fileattrib(name,attribs)` sets the specified attributes for the named file or folder.

`fileattrib(name,attribs,users)` sets the file or folder attributes for the specified subset of users.

`fileattrib(name,attribs,users,'s')` sets the specified attributes for the specified users for the contents of the named folder.

`[status,message,messageid] = fileattrib(name,attribs,___)` sets the specified file attributes and gets the function status:

- If `status` is `0`, then `message` is the error message, and `messageid` is the error message identifier.
- If `status` is `1`, then `message` is a structure containing the attributes of the named file or folder, and `messageid` is an empty string.

`[status,message] = fileattrib(name)` gets `status` and the last *successfully* set attribute structure values for the named file or folder and returns the structure to `message`. (`status` is always `1`.)

Input Arguments

name - File or folder name

The absolute or relative path for a folder or file, specified as a string. To specify all names beginning with certain characters, add the wildcard character, `*`.

Example: `fileattrib('myfile.m')`

attribs - File or folder attribute values

`'a' | 'h' | 's' | 'w' | 'x'`

File or folder attribute values, specified as a space delimited string. Use the plus (+) qualifier before an attribute to set it, and the minus (-) qualifier before an attribute to clear it.

attribvalue	Description
'a'	Archive (Microsoft Windows platform only).
'h'	Hidden file (Windows platform only).
's'	System file (Windows platform only).
'w'	Write access (Windows and UNIX platforms). Results differ by platform and application. For example, even though `fileattrib` disables the "write" privilege for a folder, making it read only, files in the folder could be writable for some platforms or applications.
'x'	Executable (UNIX platform only).

Example: `fileattrib('myfile.m', '+w -h')`

users - Subset of users

`'a' | 'g' | 'o' | 'u' | ''`

Subset of users (on UNIX platforms only), specified as a string. For all platforms other than UNIX, specify the `users` argument as an empty string, `''`. This value is not returned by `fileattrib` get operations.

Value for UNIX Users	Description
'a'	All users on UNIX platforms
'g'	Group of users
'o'	All other users
'u'	Current user

Example: `fileattrib('D:/work/results','-w','a')`

Output Arguments

status - Indication of whether attempt to set attribute was successful

`0 | 1`

If attempt to set attribute was unsuccessful, status is 0. Otherwise, status is 1.

message - Attribute structure or error message

Attribute structure or error message, depending on whether you are setting or getting attributes and status.

Getting or Setting Attributes	Status	Message contents
Setting	0	Error message
Setting	1	Empty string
Getting	1	Structure containing file attributes and values

When you are getting file attributes, the structure contains these fields and possible values.

Field name	Possible Values
Name	String containing name of file or folder
archive	0 (not set), 1 (set), or NaN (not applicable)
system	0, 1, or NaN
hidden	0, 1, or NaN
directory	0, 1, or NaN
UserRead	0, 1, or NaN
UserWrite	0, 1, or NaN
UserExecute	0, 1, or NaN
GroupRead	0, 1, or NaN

Field name	Possible Values
`GroupWrite`	`0`, `1`, or `NaN`
`GroupExecute`	`0`, `1`, or `NaN`
`OtherRead`	`0`, `1`, or `NaN`
`OtherWrite`	`0`, `1`, or `NaN`
`OtherExecute`	`0`, `1`, or `NaN`

messageid - Error message identifier

Error message identifier returned when attempt to set attribute is unsuccessful (`status` is 0). If `status` is 1, `messageid` is an empty string.

Tips

- `fileattrib` is like the DOS `attrib` command, or the UNIX `chmod` command. [2]

Examples

View Current Folder Attributes

View attributes of the current folder, assuming the current folder is `C:\my_MATLAB_files`.

```
fileattrib
```

```
ans = 

            Name: 'C:\my_MATLAB_files'
         archive: 0
          system: 0
          hidden: 0
       directory: 1
        UserRead: 1
       UserWrite: 1
     UserExecute: 1
       GroupRead: NaN
      GroupWrite: NaN
    GroupExecute: NaN
       OtherRead: NaN
      OtherWrite: NaN
    OtherExecute: NaN
```

The attributes indicate that you have read, write, and execute permissions for the current folder.

View File Attributes

View attributes of file `collatz.m`.

```
fileattrib('collatz.m')
```

2. UNIX is a registered trademark of The Open Group in the United States and other countries.

```
ans =

           Name: 'C:\my_MATLAB_files\collatz.m'
        archive: 1
         system: 0
         hidden: 0
      directory: 0
       UserRead: 1
      UserWrite: 0
    UserExecute: 1
      GroupRead: NaN
     GroupWrite: NaN
   GroupExecute: NaN
      OtherRead: NaN
     OtherWrite: NaN
   OtherExecute: NaN
```

The attributes indicate that the specfieid item is a file. You can read and execute the file, but cannot update it. The file is archived.

View Folder Attributes on a Windows System

View attributes for the folder `C:\my_MATLAB_files\doc`.

```
fileattrib('C:\my_MATLAB_files\doc')
```

```
ans =

           Name: 'C:\my_MATLAB_files\doc'
        archive: 0
         system: 0
         hidden: 0
      directory: 1
       UserRead: 1
      UserWrite: 1
    UserExecute: 1
      GroupRead: NaN
     GroupWrite: NaN
```

```
    GroupExecute: NaN
       OtherRead: NaN
      OtherWrite: NaN
    OtherExecute: NaN
```

The attributes indicate that you have read, write, and execute permissions for the specified folder.

View Folder Attributes on a UNIX System

View attributes for the folder `/public` on a UNIX system.

```
fileattrib('/public')
```

```
ans =

            Name: '/public'
         archive: NaN
          system: NaN
          hidden: NaN
       directory: 1
        UserRead: 1
       UserWrite: 1
     UserExecute: 1
       GroupRead: 1
      GroupWrite: 0
    GroupExecute: 1
       OtherRead: 1
      OtherWrite: 0
    OtherExecute: 1
```

The attributes indicate that you have read, write, and execute permissions for the specfied folder. In addition, users in your UNIX group and all others have read and execute permissions for the specified folder, but not write permissions.

Set File Attributes

Make `myfile.m` writeable.

```
fileattrib('myfile.m','+w')
```

Set File Attributes for Specific Users on UNIX

Make the folder `/home/work/results` a read-only folder for *all users* on UNIX platforms.

```
fileattrib('/home/work/results','-w','a')
```

The minus (-) preceding the write attribute, `w`, removes the write status.

Set Attributes for Folder and Its Contents

On Windows platforms, make the folder `D:\work\results` and all its contents read only and hidden.

```
fileattrib('D:\work\results','+h -w','','s')
```

Because a value for the `users` argument is not applicable on Windows systems, the argument is an empty string. The `s` argument applies the hidden and read-only attributes to the contents of the folder.

Get Attributes Structure for a Folder

Get the attributes for the folder `results` and return them to a structure:

```
[stat,struc]=fileattrib('results')

stat =
     1

struc =
            Name: 'D:\work\results'
         archive: 0
          system: 0
          hidden: 0
       directory: 1
        UserRead: 1
       UserWrite: 1
     UserExecute: 1
       GroupRead: NaN
```

```
      GroupWrite: NaN
    GroupExecute: NaN
       OtherRead: NaN
      OtherWrite: NaN
    OtherExecute: NaN
```

The operation is successful as indicated by the status, `stat`, value of 1. The structure, `struc`, contains the file attributes.

Access the name attribute value in the structure. MATLAB returns the path for `results`.

```
struc.Name

ans =
D:\work\results
```

Get Attributes Structure for Multiple Files

Get the attributes for all files in the current folder with names that begin with `new`.

```
[stat,struc]=fileattrib('new*')

stat =
     1

mess =
1x3 struct array with fields:
    Name
    archive
    system
    hidden
    directory
    UserRead
    UserWrite
    UserExecute
    GroupRead
    GroupWrite
```

```
    GroupExecute
    OtherRead
    OtherWrite
    OtherExecute
```

The results indicate there are three matching files.

View the file names.

```
struc.Name
```

```
ans =
D:\work\results\newname.m

ans =
D:\work\results\newone.m

ans =
D:\work\results\newtest.m
```

View just the second file name.

```
struct(2).Name
```

```
ans =
D:\work\results\newname.m
```

Successfully Set Attributes for a File and Get Messages

Show output that results when an attempt to set file attributes is successful.

```
[status, message, messageid] = fileattrib('C:/my_MATLAB_files\doc','+h
```

```
status =

     1


message =
```

```
     ''


messageid =

     ''
```

The `status` value of `1` indicates the set operation was successful; therefore, no error `message` or `messageid` is returned.

Unsuccessfully Set Attributes for a File and Get Messages

Show output that results when an attempt to set file attributes is unsuccessful.

```
[status, message, messageid] = fileattrib('C:/my_MATLAB_files\doc','+h w-
```

```
status =

     0


message =

Illegal file mode characters on the current platform.


messageid =

MATLAB:FILEATTRIB:ModeSyntaxError
```

The `status` value of `0` indicates the set operation was unsuccessful. The minus sign incorrectly appears after `w`, instead of before it.

See Also

`cd` | `copyfile` | `delete` | `dir` | `ls` | `mkdir` | `movefile` | `rmdir`

Purpose

Open Current Folder browser, or select it if already open

Syntax

```
filebrowser
```

Description

`filebrowser` opens the Current Folder browser, or if it is already open, makes it the selected tool.

See Also

`cd` | `copyfile` | `fileattrib` | `ls` | `mkdir` | `movefile` | `pwd` | `rmdir`

How To

- "Working with Files and Folders"

filemarker

Purpose

Character to separate file name and internal function name

Syntax

```
M = filemarker
```

Description

`M = filemarker` returns the character that separates a file and a within-file function name.

Examples

On the Microsoft Windows platform, for example, `filemarker` returns the '>' character:

```
filemarker

ans =
   >
```

You can use the following command on any platform to get the help text for the local function `pdeodes` defined in `pdepe.m`:

```
 helptext = help(['imwrite' filemarker 'validateSizes'])

helptext =
  How many bytes does each element occupy in memory?
```

You can use the `filemarker` character to indicate a location within a MATLAB program file where you want to set a breakpoint, for example. On all platforms, if you need to distinguish between two nested functions with the same name, use the forward slash (/) character to indicate the path to a particular instance of a function.

For instance, suppose `myfile.m` contains the following code:

```
function x = A(p1, p2)
...
  function y = B(p3)
    ...
  end
  function m = C(p4)
    ...
```

```
    end
end

function z = C(p5)
...
  function y = D(p6)
    ...
  end
end
```

To indicate that you want to set a breakpoint at `function y` nested within `function x`, use the following command on the Windows platform:

```
dbstop myfile>x/y
```

To indicate that you want to set a breakpoint at `function m` nested within `function x` use the following command on the Windows platform:

```
dbstop myfile>m
```

In the first case, you specify `x/y` because `myfile.m` contains two nested functions named `y`. In the second case, there is no need to specify `x/m` because there is only one function `m` within `myfile.m`.

See Also

`filesep`

fileparts

Purpose

Parts of file name and path

Note The fourth output argument of `fileparts` (file version) is no longer supported and has been removed. Calling the function with more than three output arguments generates an error.

Syntax

```
[pathstr, name, ext] = fileparts(filename)
```

Description

`[pathstr, name, ext] = fileparts(filename)` returns the path name, file name, and extension for the specified file. The file does not have to exist. `filename` is a string enclosed in single quotes. The returned `ext` field contains a dot (`.`) before the file extension.

Tips

- `fileparts` only parses file names. It does not verify that a file or a folder exists.
- You can reconstruct the file from the parts using:

  ```
  fullfile(pathstr,[name ext])
  ```

- On Microsoft Windows systems, you can use either forward (`/`) or back (`\`) slashes as path delimiters, even within the same string. On UNIX and Macintosh systems, use only `/` as a delimiter. You can use the `filesep` function to insert the correct separator character for the platform on which your code executes:

  ```
  sep = filesep;
  file = ['H:' sep 'user4' sep 'matlab' sep 'myfile.txt'];

  file =
  H:\user4\matlab\myfile.txt
  ```

- If the input consists of a folder name only, be sure that the right-most character is a delimiter (`/` or `\`). Otherwise, `fileparts` parses the trailing portion of `filename` as the name of a file and returns it in `name` instead of in `pathstr`.

Input Arguments

filename

String containing a name of a file or folder, which can include a path and file extension. The function interprets all characters following the right-most delimiter as a file name plus extension.

Output Arguments

pathstr

String containing the part of `filename` interpreted as a path name

name

String containing the name of the file without any extension

ext

String containing the file extension only, beginning with a period (`.`)

Definitions

Path Name

The full or partial path to a destination folder location, always the initial portion of the filename string. Path names end with a slash character and, where appropriate, can begin with a drive letter. Windows paths use backward slashes (`\`). UNIX and Macintosh paths use forward slashes (`/`).

Examples

Return the pieces of a file specification string to the separate string outputs `pathstr`, `name`, and `ext`. The full file specification is:

```
file = 'H:\user4\matlab\myfile.txt';
[pathstr, name, ext] = fileparts(file)

pathstr =
H:\user4\matlab

name =
myfile

ext =
```

```
.txt
```

Query parts of a user `.cshrc` file:

```
[p,n,e] = fileparts('/home/jsmith/.cshrc')

pathstr =
/home/jsmith

name =
   Empty string: 1-by-0

ext =
.cshrc
```

`fileparts` interprets the entire file name as an extension because it begins with a period.

Alternatives

Use `uigetfile` to interactively select and return a file name and path, or `uigetdir` to interactively select and return a path name. If you call `fileparts` with the output of `uigetfile`, you can parse out the file name and extension.

See Also

`filesep` | `fullfile` | `pathsep` | `uigetdir` | `uigetfile`

Tutorials

- “Path Names in MATLAB”

Purpose

Read contents of file into string

Syntax

```
text = fileread(filename)
```

Description

text = fileread(*filename*) returns the contents of the file *filename* as a MATLAB string.

Examples

Read and search the file Contents.m in the MATLAB iofun directory for the reference to fileread:

```
% find the correct directory and file
io_contents = ...
  fullfile(matlabroot, 'toolbox', 'matlab', 'iofun', 'Contents.m');

% read the file
filetext = fileread(io_contents);

% search for the line of code that includes 'fileread'
% each line is separated by a newline ('\n')

expr = '[^\n]*fileread[^\n]*';
fileread_info = regexp(filetext, expr, 'match');
```

See Also

fgetl | fgets | fscanf | fread | importdata | textscan | type

filesep

Purpose

File separator for current platform

Syntax

```
f = filesep
```

Description

`f = filesep` returns the platform-specific file separator character. The file separator is the character that separates individual folder and file names in a path string.

Examples

Create a path to the `iofun` folder on a Microsoft Windows platform:

```
iofun_dir = ['toolbox' filesep 'matlab' filesep 'iofun']

iofun_dir =
   toolbox\matlab\iofun
```

Create a path to `iofun` on a UNIX[3] platform.

```
iodir = ['toolbox' filesep 'matlab' filesep 'iofun']

iodir =
   toolbox/matlab/iofun
```

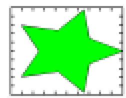
See Also

`fullfile` | `fileparts` | `pathsep`

3. UNIX is a registered trademark of The Open Group in the United States and other countries.

Purpose

Filled 2-D polygons



Syntax

```
fill(X,Y,C)
fill(X,Y,ColorSpec)
fill(X1,Y1,C1,X2,Y2,C2,...)
fill(...,'PropertyName',PropertyValue)
h = fill(...)
```

Description

The `fill` function creates colored polygons.

`fill(X,Y,C)` creates filled polygons from the data in `X` and `Y` with vertex color specified by `C`. `C` is a vector or matrix used as an index into the colormap. If `C` is a row vector, `length(C)` must equal `size(X,2)` and `size(Y,2)`; if `C` is a column vector, `length(C)` must equal `size(X,1)` and `size(Y,1)`. If necessary, `fill` closes the polygon by connecting the last vertex to the first.

`fill(X,Y,`*`ColorSpec`*`)` fills two-dimensional polygons specified by `X` and `Y` with the color specified by `ColorSpec`.

`fill(X1,Y1,C1,X2,Y2,C2,...)` specifies multiple two-dimensional filled areas.

`fill(...,'`*`PropertyName`*`',PropertyValue)` allows you to specify property names and values for a patch graphics object.

`h = fill(...)` returns a vector of handles to patch graphics objects, one handle per patch object.

Tips

If `X` or `Y` is a matrix, and the other is a column vector with the same number of elements as rows in the matrix, `fill` replicates the column vector argument to produce a matrix of the required size. `fill` forms a vertex from corresponding elements in `X` and `Y` and creates one polygon from the data in each column.

The type of color shading depends on how you specify color in the argument list. If you specify color using `ColorSpec`, `fill` generates flat-shaded polygons by setting the patch object's `FaceColor` property to the corresponding RGB triple.

If you specify color using `C`, `fill` scales the elements of `C` by the values specified by the axes property `CLim`. After scaling `C`, `C` indexes the current colormap.

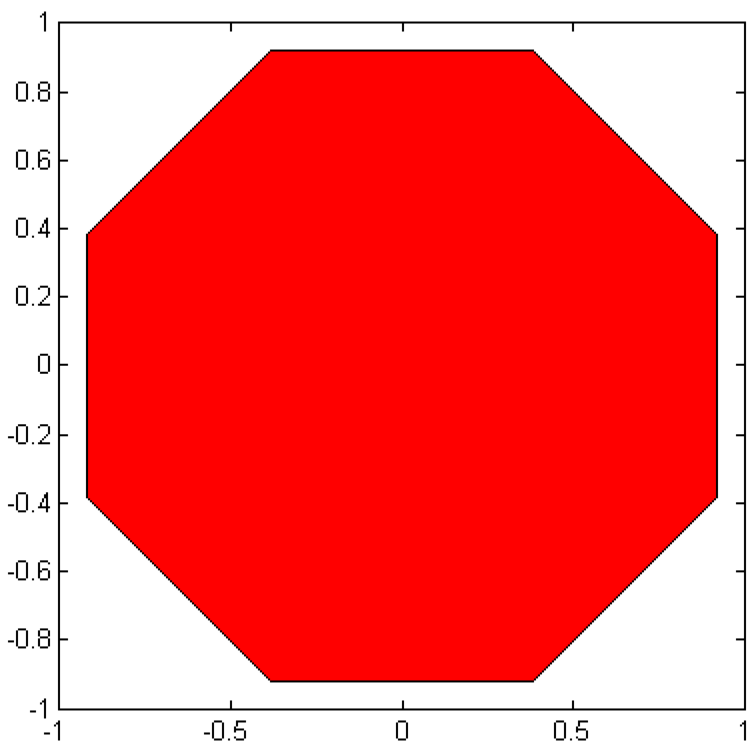
If `C` is a row vector, `fill` generates flat-shaded polygons where each element determines the color of the polygon defined by the respective column of the `X` and `Y` matrices. Each patch object's `FaceColor` property is set to `'flat'`. Each row element becomes the `CData` property value for the `n`th patch object, where *n* is the corresponding column in `X` or `Y`.

If `C` is a column vector or a matrix, `fill` uses a linear interpolation of the vertex colors to generate polygons with interpolated colors. It sets the patch graphics object `FaceColor` property to `'interp'` and the elements in one column become the `CData` property value for the respective patch object. If `C` is a column vector, `fill` replicates the column vector to produce the required sized matrix.

Examples

Create a red octagon.

```
t = (1/16:1/8:1)'*2*pi;
x = sin(t);
y = cos(t);
fill(x,y,'r')
axis square
```



See Also

`axis` | `caxis` | `colormap` | `ColorSpec` | `fill3` | `patch`

Purpose

Filled 3-D polygons

Syntax

```
fill3(X,Y,Z,C)
fill3(X,Y,Z,ColorSpec)
fill3(X1,Y1,Z1,C1,X2,Y2,Z2,C2,...)
fill3(...,'PropertyName',PropertyValue)
h = fill3(...)
```

Description

The `fill3` function creates flat-shaded and Gouraud-shaded polygons.

`fill3(X,Y,Z,C)` fills three-dimensional polygons. `X`, `Y`, and `Z` triplets specify the polygon vertices. If `X`, `Y`, or `Z` is a matrix, `fill3` creates *n* polygons, where *n* is the number of columns in the matrix. `fill3` closes the polygons by connecting the last vertex to the first when necessary.

`C` specifies color, where `C` is a vector or matrix of indices into the current colormap. If `C` is a row vector, `length(C)` must equal `size(X,2)` and `size(Y,2)`; if `C` is a column vector, `length(C)` must equal `size(X,1)` and `size(Y,1)`.

`fill3(X,Y,Z,`*`ColorSpec`*`)` fills three-dimensional polygons defined by `X`, `Y`, and `Z` with color specified by `ColorSpec`.

`fill3(X1,Y1,Z1,C1,X2,Y2,Z2,C2,...)` specifies multiple filled three-dimensional areas.

`fill3(...,'`*`PropertyName`*`',PropertyValue)` allows you to set values for specific patch properties.

`h = fill3(...)` returns a vector of handles to patch graphics objects, one handle per patch.

Algorithms

If `X`, `Y`, and `Z` are matrices of the same size, `fill3` forms a vertex from the corresponding elements of `X`, `Y`, and `Z` (all from the same matrix location), and creates one polygon from the data in each column.

If X, Y, or Z is a matrix, fill3 replicates any column vector argument to produce matrices of the required size.

If you specify color using ColorSpec, fill3 generates flat-shaded polygons and sets the patch object FaceColor property to an RGB triple.

If you specify color using C, fill3 scales the elements of C by the axes property CLim, which specifies the color axis scaling parameters, before indexing the current colormap.

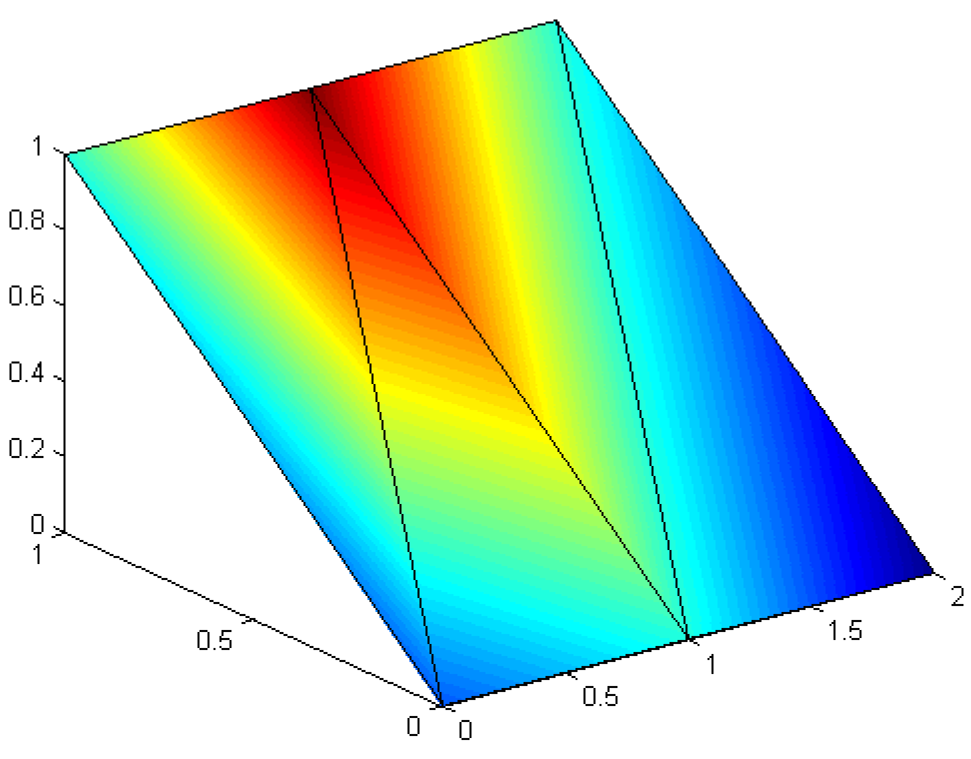
If C is a row vector, fill3 generates flat-shaded polygons and sets the FaceColor property of the patch objects to 'flat'. Each element becomes the CData property value for the respective patch object.

If C is a column vector or a matrix, fill3 generates polygons with interpolated colors and sets the patch object FaceColor property to 'interp'. fill3 uses a linear interpolation of the vertex colormap indices when generating polygons with interpolated colors. The elements in one column become the CData property value for the respective patch object. If C is a column vector, fill3 replicates the column vector to produce the required sized matrix.

Examples

Create four triangles with interpolated colors.

```
X = [0 1 1 2;1 1 2 2;0 0 1 1];
Y = [1 1 1 1;1 0 1 0;0 0 0 0];
Z = [1 1 1 1;1 0 1 0;0 0 0 0];
C = [0.5000 1.0000 1.0000 0.5000;
     1.0000 0.5000 0.5000 0.1667;
     0.3330 0.3330 0.5000 0.5000];
fill3(X,Y,Z,C)
```



See Also

axis | caxis | colormap | ColorSpec | fill | patch

Purpose

1-D digital filter

Syntax

```
y = filter(b,a,X)
[y,zf] = filter(b,a,X)
[y,zf] = filter(b,a,X,zi)
y = filter(b,a,X,zi,dim)
[...] = filter(b,a,X,[],dim)
```

Description

The `filter` function filters a data sequence using a digital filter which works for both real and complex inputs. The filter is a *direct form II transposed* implementation of the standard difference equation (see "Algorithm").

`y = filter(b,a,X)` filters the data in vector `X` with the filter described by numerator coefficient vector `b` and denominator coefficient vector `a`. If `a(1)` is not equal to `1`, `filter` normalizes the filter coefficients by `a(1)`. If `a(1)` equals `0`, `filter` returns an error.

If `X` is a matrix, `filter` operates on the columns of `X`. If `X` is a multidimensional array, `filter` operates on the first nonsingleton dimension.

`[y,zf] = filter(b,a,X)` returns the final conditions, `zf`, of the filter delays. If `X` is a row or column vector, output `zf` is a column vector of `max(length(a),length(b))-1`. If `X` is a matrix, `zf` is an array of such vectors, one for each column of `X`, and similarly for multidimensional arrays.

`[y,zf] = filter(b,a,X,zi)` accepts initial conditions, `zi`, and returns the final conditions, `zf`, of the filter delays. Input `zi` is a vector of length `max(length(a),length(b))-1`, or an array with the leading dimension of size `max(length(a),length(b))-1` and with remaining dimensions matching those of `X`.

`y = filter(b,a,X,zi,dim)` and `[...]  = filter(b,a,X,[],dim)` operate across the dimension `dim`.

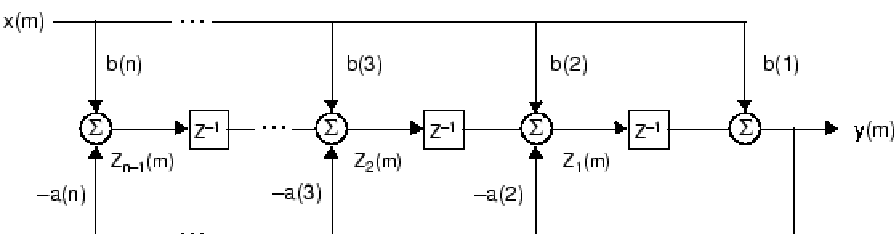
Examples

You can use `filter` to find a running average without using a `for` loop. This example finds the running average of a 16-element vector, using a window size of 5.

```
data = [1:0.2:4]';
windowSize = 5;
filter(ones(1,windowSize)/windowSize,1,data)

ans =
    0.2000
    0.4400
    0.7200
    1.0400
    1.4000
    1.6000
    1.8000
    2.0000
    2.2000
    2.4000
    2.6000
    2.8000
    3.0000
    3.2000
    3.4000
    3.6000
```

Algorithms

The `filter` function is implemented as a direct form II transposed structure,



or

```
y(n) = b(1)*x(n) + b(2)*x(n-1) + ... + b(nb+1)*x(n-nb)
                 - a(2)*y(n-1) - ... - a(na+1)*y(n-na)
```

where `n-1` is the filter order, which handles both FIR and IIR filters [1], `na` is the feedback filter order, and `nb` is the feedforward filter order.

The operation of `filter` at sample m is given by the time domain difference equations

$$
\begin{aligned}
y(m) &= b(1)x(m) + z_1(m-1) \\
z_1(m) &= b(2)x(m) + z_2(m-1) - a(2)y(m) \\
\vdots \quad &= \quad \vdots \qquad\qquad \vdots \\
z_{n-2}(m) &= b(n-1)x(m) + z_{n-1}(m-1) - a(n-1)y(m) \\
z_{n-1}(m) &= b(n)x(m) - a(n)y(m)
\end{aligned}
$$

The input-output description of this filtering operation in the z-transform domain is a rational transfer function,

$$
Y(z) = \frac{b(1) + b(2)z^{-1} + \ldots + b(nb+1)z^{-nb}}{1 + a(2)z^{-1} + \ldots + a(na+1)z^{-na}} X(z)
$$

References

[1] Oppenheim, A. V. and R.W. Schafer. *Discrete-Time Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, 1989, pp. 311-312.

See Also

`filter2` | `filtfilt` | `filtic`

filter2

Purpose

2-D digital filter

Syntax

```
Y = filter2(h,X)
Y = filter2(h,X,shape)
```

Description

`Y = filter2(h,X)` filters the data in `X` with the two-dimensional FIR filter in the matrix `h`. It computes the result, `Y`, using two-dimensional correlation, and returns the central part of the correlation that is the same size as `X`.

`Y = filter2(h,X,`*`shape`*`)` returns the part of `Y` specified by the `shape` parameter. *`shape`* is a string with one of these values:

'full'	Returns the full two-dimensional correlation. In this case, `Y` is larger than `X`.
'same'	(default) Returns the central part of the correlation. In this case, `Y` is the same size as `X`.
'valid'	Returns only those parts of the correlation that are computed without zero-padded edges. In this case, `Y` is smaller than `X`.

Tips

Two-dimensional correlation is equivalent to two-dimensional convolution with the filter matrix rotated 180 degrees. See the Algorithm section for more information about how `filter2` performs linear filtering.

Algorithms

Given a matrix `X` and a two-dimensional FIR filter `h`, `filter2` rotates your filter matrix 180 degrees to create a convolution kernel. It then calls `conv2`, the two-dimensional convolution function, to implement the filtering operation.

`filter2` uses `conv2` to compute the full two-dimensional convolution of the FIR filter with the input matrix. By default, `filter2` then extracts the central part of the convolution that is the same size as the input

matrix, and returns this as the result. If the `shape` parameter specifies an alternate part of the convolution for the result, `filter2` returns the appropriate part.

See Also

`conv2` | `filter`

find

Purpose

Find indices and values of nonzero elements

Syntax

```
ind = find(X)
ind = find(X, k)
ind = find(X, k, 'first')
ind = find(X, k, 'last')
[row,col] = find(X, ...)
[row,col,v] = find(X, ...)
```

Description

`ind = find(X)` locates all nonzero elements of array `X`, and returns the linear indices of those elements in vector `ind`. If `X` is a row vector, then `ind` is a row vector; otherwise, `ind` is a column vector. If `X` contains no nonzero elements or is an empty array, then `ind` is an empty array.

`ind = find(X, k)` or `ind = find(X, k, 'first')` returns at most the first `k` indices corresponding to the nonzero entries of `X`. `k` must be a positive integer, but it can be of any numeric data type.

`ind = find(X, k, 'last')` returns at most the last `k` indices corresponding to the nonzero entries of `X`.

`[row,col] = find(X, ...)` returns the row and column indices of the nonzero entries in the matrix `X`. This syntax is especially useful when working with sparse matrices. If `X` is an N-dimensional array with N > 2, `col` contains linear indices for the columns. For example, for a 5-by-7-by-3 array `X` with a nonzero element at `X(4,2,3)`, `find` returns 4 in `row` and 16 in `col`. That is, (7 columns in page 1) + (7 columns in page 2) + (2 columns in page 3) = 16.

`[row,col,v] = find(X, ...)` returns a column or row vector `v` of the nonzero entries in `X`, as well as row and column indices. If `X` is a logical expression, then `v` is a logical array. Output `v` contains the non-zero elements of the logical array obtained by evaluating the expression `X`. For example,

```
A= magic(4)
A =
    16     2     3    13
     5    11    10     8
```

```
     9     7     6    12
     4    14    15     1

[r,c,v]= find(A>10);

r', c', v'
ans =
     1     2     4     4     1     3
ans =
     1     2     2     3     4     4
ans =
     1     1     1     1     1     1
```

Here the returned vector `v` is a logical array that contains the nonzero elements of `N` where

```
N=(A>10)
```

Examples

Example 1

```
X = [1 0 4 -3 0 0 0 8 6];
indices = find(X)
```

returns linear indices for the nonzero entries of `X`.

```
indices =
     1     3     4     8     9
```

Example 2

You can use a logical expression to define `X`. For example,

```
find(X > 2)
```

returns linear indices corresponding to the entries of `X` that are greater than 2.

```
ans =
     3     8     9
```

Example 3

The following `find` command

```
X = [3 2 0; -5 0 7; 0 0 1];
[r,c,v] = find(X)
```

returns a vector of row indices of the nonzero entries of `X`

```
r =
     1
     2
     1
     2
     3
```

a vector of column indices of the nonzero entries of `X`

```
c =
     1
     1
     2
     3
     3
```

and a vector containing the nonzero entries of `X`.

```
v =
     3
    -5
     2
     7
     1
```

Example 4

The expression

```
X = [3 2 0; -5 0 7; 0 0 1];
[r,c,v] = find(X>2)
```

returns a vector of row indices of the nonzero entries of `N` where `N=(X>2)`

```
r =
     1
     2
```

a vector of column indices of the nonzero entries of `N` where `N=(X>2)`

```
c =
     1
     3
```

and a logical array that contains the nonzero elements of `N` where `N=(X>2)`.

```
v =
     1
     1
```

Recall that when you use `find` on a logical expression, the output vector `v` does not contain the nonzero entries of the input array. Instead, it contains the nonzero values returned after evaluating the logical expression.

Example 5

Some operations on a vector

```
x = [11  0  33  0  55]';

find(x)
ans =
     1
     3
     5

find(x == 0)
ans =
     2
```

```
     4

find(0 < x & x < 10*pi)
ans =
     1
```

Example 6

For the matrix

```
M = magic(3)
M =
     8     1     6
     3     5     7
     4     9     2

find(M > 3, 4)
```

returns the indices of the first four entries of `M` that are greater than 3.

```
ans =
     1
     3
     5
     6
```

Example 7

If `X` is a vector of all zeros, `find(X)` returns an empty matrix. For example,

```
indices = find([0;0;0])
indices =
   Empty matrix: 0-by-1
```

See Also

`nonzeros` | `sparse` | `colon` | `ind2sub` | Logical Operators: Elementwise | Logical Operators: Short-circuit | Relational Operators

Purpose

Find all graphics objects

Syntax

```
object_handles = findall(handle_list)
object_handles = findall(handle_list,'property','value',...)
```

Description

`object_handles = findall(handle_list)` returns the handles, including hidden handles, of all objects in the hierarchy under the objects identified in `handle_list`.

`object_handles = findall(handle_list,'property','value',...)` returns the handles of all objects in the hierarchy under the objects identified in `handle_list` that have the specified properties set to the specified values.

Tips

`findall` is similar to `findobj`, except that it finds objects even if their `HandleVisibility` is set to `off`.

Examples

```
plot(1:10)
xlabel xlab
a = findall(gcf)
b = findobj(gcf)
c = findall(b,'Type','text') % return the xlabel handle twice
d = findobj(b,'Type','text') % can't find the xlabel handle
```

See Also

`allchild` | `findobj`

findfigs

Purpose Find visible offscreen figures

Syntax

```
findfigs
```

Description

`findfigs` finds all visible figure windows whose display area is off the screen and positions them on the screen.

A window appears to the MATLAB software to be offscreen when its display area (the area not covered by the window's title bar, menu bar, and toolbar) does not appear on the screen.

This function is useful when you are bringing an application from a larger monitor to a smaller one (or one with lower resolution). Windows visible on the larger monitor may appear offscreen on a smaller monitor. Using `findfigs` ensures that all windows appear on the screen.

Purpose

Locate graphics objects with specific properties

Syntax

```
findobj
h = findobj
h = findobj('PropertyName',PropertyValue,...)
h =
findobj('PropertyName',PropertyValue,'-logicaloperator',
   PropertyName',PropertyValue,...)
h = findobj('-regexp','PropertyName','regexp',...)
h = findobj('-property','PropertyName')
h = findobj(objhandles,...)
h = findobj(objhandles,'-depth',d,...)
h = findobj(objhandles,'flat','PropertyName',PropertyValue,
   ...)
```

Description

`findobj` returns handles of the root object and all its descendants without assigning the result to a variable.

`h = findobj` returns handles of the root object and all its descendants.

`h = findobj('`*`PropertyName`*`',PropertyValue,...)` returns handles of all graphics objects having the property *`PropertyName`*, set to the value `PropertyValue`. You can specify more than one property/value pair, in which case, `findobj` returns only those objects having all specified values.

`h =`
`findobj('`*`PropertyName`*`',PropertyValue,'`*`-logicaloperator`*`',`
`PropertyName``',PropertyValue,...)` applies the logical operator to the property value matching. Possible values for *`-logicaloperator`* are:

- `-and`
- `-or`
- `-xor`
- `-not`

See "Logical Operators" for an explanation of logical operators.

`h = findobj('-regexp','PropertyName','regexp',...)` matches objects using regular expressions as if the value of you passed the property `PropertyName` to the `regexp` function as

```
regexp(PropertyValue,'regexp')
```

If a match occurs, `findobj` returns the object handle. See the `regexp` function for information on how the MATLAB software uses regular expressions.

`h = findobj('-property','PropertyName')` finds all objects having the specified property.

`h = findobj(objhandles,...)` restricts the search to objects listed in `objhandles` and their descendants.

`h = findobj(objhandles,'-depth',d,...)` specifies the depth of the search. The depth argument `d` controls how many levels under the handles in `objhandles` MATLAB traverses. Specify `d` as `inf` to get the default behavior of all levels. Specify `d` as `0` to get the same behavior as using the `flat` argument.

`h = findobj(objhandles,'flat','PropertyName',PropertyValue,...)` restricts the search to those objects listed in `objhandles` and does not search descendants.

`findobj` returns an error if a handle refers to a nonexistent graphics object.

`findobj` correctly matches any legal property value. For example,

```
findobj('Color','r')
```

finds all objects having a `Color` property set to `red`, `r`, or `[1 0 0]`.

When a graphics object is a descendant of more than one object identified in `objhandles`, MATLAB searches the object each time `findobj` encounters its handle. Therefore, implicit references to a graphics object can result in multiple returns of its handle.

To find handle objects that meet specified conditions, use `handle.findobj`.

Examples

Find all line objects in the current axes:

```
h = findobj(gca,'Type','line')
```

Find all objects having a `Label` set to `'foo'` and a `String` set to `'bar'`:

```
h = findobj('Label','foo','-and','String','bar');
```

Find all objects whose `String` is not `'foo'` and is not `'bar'`:

```
h = findobj('-not','String','foo','-not','String','bar');
```

Find all objects having a `String` set to `'foo'` and a `Tag` set to `'button one'` and whose `Color` is not `'red'` or `'blue'`:

```
h = findobj('String','foo','-and','Tag','button one',...
 '-and','-not',{'Color','red','-or','Color','blue'})
```

Find all objects for which you have assigned a value to the `Tag` property (that is, the value is not the empty string `''`):

```
h = findobj('-regexp','Tag','[^'']')
```

Find all children of the current figure that have their `BackgroundColor` property set to a certain shade of gray (`[.7 .7 .7]`). This statement also searches the current figure for the matching property value pair.

```
h = findobj(gcf,'-depth',1,'BackgroundColor',[.7 .7 .7])
```

See Also

`copyobj` | `findall` | `handle.findobj` | `gcf` | `gca` | `gcbo` | `gco` | `get` | `regexp` | `set`

Tutorials

- "Example — Using Logical Operators and Regular Expression"

Purpose

Find handle objects matching specified conditions

Syntax

```
Hmatch = findobj(Hobj,<conditions>)
Hmatch = findobj(Hobj,'-property','PropertyName')
```

Description

`Hmatch = findobj(Hobj,<conditions>)` finds `handle` class objects that meet the specified conditions. Specify conditions as property name/property value pairs, indicating that the objects you are trying to find have specific property values.

The `Hobj` argument must be an array of `handle` objects. The returned value, `Hmatch` contains an array of object `handles` that match the conditions.

`Hmatch = findobj(Hobj,'-property','PropertyName')` finds all object in `Hobj` having the specified property.

`findobj` has access only to public members of the objects in `Hobj`. `findobj` can find dynamic properties. See "Find Handle Objects" for more information on using `findobj`.

You cannot use regular expression with `handle` class objects.

Examples

Find the object with a specific property value. Given the `handle` class, `BasicHandle`:

```
classdef BasicHandle < handle
   properties
      Prop1
   end
   methods
      function obj = BasicHandle(val)
         if nargin > 0
            obj.Prop1 = val;
         end
      end
   end
end
```

Create an array of BasicHandle objects:

```
h(1) = BasicHandle(7);
h(2) = BasicHandle(11);
h(3) = BasicHandle(27);
```

Find the handle of the object whose Prop1 property has a value of 7:

```
h7 = findobj(h,'Prop1',7);
h7.Prop1

ans =

     7
```

Find the object with a specific dynamic property. Given the button class:

```
classdef button < dynamicprops
   properties
      UiHandle
   end
   methods
      function obj = button(pos)
         if nargin > 0
            if length(pos) == 4
               obj.UiHandle = uicontrol('Position',pos,...
                'Style','pushbutton');
            else
               error('Improper position')
            end
         end
      end
   end
end
```

Create an array of button objects, only one element of which defines a dynamic property. Use `findobj` to get the handle of the object with the dynamic property named `ButtonCoord`:

```
b(1) = button([20 40 80 20]);
b(1).addprop('ButtonCoord');
b(1).ButtonCoord = [2,3];
b(2) = button([120 40 80 20]);
b(3) = button([220 40 80 20]);

h =  findobj(b,'-property','ButtonCoord');
h.ButtonCoord

ans =

     2     3
```

See Also

`findprop` | `handle`

Purpose Find `meta.property` object associated with property name

Syntax

```
p = findprop(h,'Name')
```

Description `p = findprop(h,'Name')` returns the `meta.property` object associated with the property `Name` of the object `h`. `Name` can be a property defined by the class of `h` or a dynamic property defined only for the object `h`.

Examples Use `findprop` to view property attribute settings:

```
findprop(containers.Map,'Count')

ans =

  meta.property handle
  Package: meta

  Properties:
                   Name: 'Count'
            Description: 'Number of pairs in the collection'
    DetailedDescription: ''
              GetAccess: 'public'
              SetAccess: 'private'
              Dependent: 1
               Constant: 0
               Abstract: 0
              Transient: 1
                 Hidden: 0
          GetObservable: 0
          SetObservable: 0
               AbortSet: 0
              GetMethod: []
              SetMethod: []
          DefiningClass: [1x1 meta.class]

  Methods, Events, Superclasses
```

See Also `handle` | `findobj (handle)` | `meta.property`

How To

- "Getting Information About Properties"
- "Getting Information About Properties"
- "Dynamic Properties — Adding Properties to an Instance"

findstr

Purpose

Find string within another, longer string

Note `findstr` is not recommended. Use `strfind` instead.

Syntax

```
k = findstr(str1, str2)
```

Description

`k = findstr(str1, str2)` searches the longer of the two input strings for any occurrences of the shorter string, returning the starting index of each such occurrence in the double array `k`. If no occurrences are found, then `findstr` returns the empty array, `[]`.

The search performed by `findstr` is case sensitive. Any leading and trailing blanks in either input string are explicitly included in the comparison.

Unlike the `strfind` function, the order of the input arguments to `findstr` is not important. This can be useful if you are not certain which of the two input strings is the longer one.

Examples

```
s = 'Find the starting indices of the shorter string.';

findstr(s, 'the')
ans =
     6    30

findstr('the', s)
ans =
     6    30
```

See Also

`strfind` | `strtok` | `strcmp` | `strncmp` | `strcmpi` | `strncmpi` | `regexp` | `regexpi` | `regexprep`

Purpose

Termination file for MATLAB program

Description

When the MATLAB program quits, it runs a script called `finish.m`, if the script exists and is on the search path MATLAB uses or in the current directory. This is a file you create yourself that instructs MATLAB to perform any final tasks just prior to terminating. For example, you might want to save the data in your workspace to a MAT-file before MATLAB exits.

`finish.m` is invoked whenever you do one of the following:

- Click the Close box ☒ in the MATLAB desktop on Microsoft Windows platforms or the equivalent on UNIX[4] platforms
- Type `quit` or `exit` at the Command Window prompt

Tips

When using Handle Graphics features in `finish.m`, use `uiwait`, `waitfor`, or `drawnow` so that figures are visible. See the reference pages for these functions for more information.

Examples

Two sample termination files are provided with MATLAB in *`matlabroot`*`/toolbox/local`.

- `finishsav.m` — Saves the workspace to a MAT-file when MATLAB quits.
- `finishdlg.m` — Displays a dialog allowing you to cancel quitting and saves the workspace. See also the "Confirmation Dialogs Preferences" and the option for exiting MATLAB.

To create a termination file, make a copy of one of these sample files, changing the name to `finish.m`, and add it to the path to use it. You can modify the file to include any operations you want the termination file to perform.

See Also

`quit` | `exit` | `startup`

How To

- "Exit MATLAB"

4. UNIX is a registered trademark of The Open Group in the United States and other countries.

Purpose

Display FITS metadata

Syntax

```
fitsdisp(filename)
fitsdisp(filename,Name,Value)
```

Description

`fitsdisp(filename)` displays metadata for all the Header/Data Units (HDUs) found in the FITS file specified by `filename`.

`fitsdisp(filename,Name,Value)` displays metadata for all the Header/Data Units (HDUs) found in the FITS file with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

filename

Text string specifying the name of an existing FITS file.

Name-Value Pair Arguments

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

'Index'

Positive scalar value or vector specifying the HDUs.

'Mode'

One of the following strings:

- `standard` – Display standard keywords
- `min` – Display only HDU types and sizes
- `full` – Display all HDU keywords

Default: `standard`

Examples

Display metadata in the 2nd HDU in the FITS file.

```
fitsdisp('tst0012.fits','Index',2);
```

Display the metadata in the 1st, 3rd, and 5th HDUs in a file.

```
fitsdisp('tst0012.fits','Index',[1 3 5]);
```

Display all metadata in the 5th HDU in a file

```
fitsdisp('tst0012.fits','Index',5,'Mode','full');
```

References

For copyright information, see the `cfitsiocopyright.txt` file.

See Also

`fitsread` | `fitswrite` | `fitsdisp` | matlab.io.fits

Purpose

Information about FITS file

Syntax

```
info = fitsinfo(filename)
```

Description

`info = fitsinfo(filename)` returns the structure, `info`, with fields that contain information about the contents of a Flexible Image Transport System (FITS) file. `filename` is a string enclosed in single quotes that specifies the name of the FITS file.

The `info` structure contains the following fields, listed in the order they appear in the structure. In addition, the info structure can also contain information about any number of optional file components, called *extensions* in FITS terminology. For more information, see "FITS File Extensions" on page 1-1758.

Field Name	Description	Return Type
`Filename`	Name of the file	String
`FileModDate`	File modification date	String
`FileSize`	Size of the file in bytes	Double
`Contents`	List of extensions in the file in the order that they occur	Cell array of strings
`PrimaryData`	Information about the primary data in the FITS file	Structure array

PrimaryData

The `PrimaryData` field is a structure that describes the primary data in the file. The following table lists the fields in the order they appear in the structure.

Field Name	Description	Return Type
`DataType`	Precision of the data	String
`Size`	Array containing the size of each dimension	Double array

Field Name	Description	Return Type
`DataSize`	Size of the primary data in bytes	Double
`MissingDataValue`	Value used to represent undefined data	Double
`Intercept`	Value, used with `Slope`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value + Intercept`	Double
`Slope`	Value, used with `Intercept`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value + Intercept`	Double
`Offset`	Number of bytes from beginning of the file to the location of the first data value	Double
`Keywords`	A number-of-keywords-by-3 cell array containing keywords, values, and comments of the header in each column	Cell array of strings

FITS File Extensions

A FITS file can also include optional extensions. If the file contains any of these extensions, the `info` structure can contain these additional fields.

- AsciiTable — Numeric information in tabular format, stored as ASCII characters

- BinaryTable — Numeric information in tabular format, stored in binary representation
- Image — A multidimensional array of pixels
- Unknown — Nonstandard extension

AsciiTable Extension

The `AsciiTable` structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
`Rows`	Number of rows in the table	Double
`RowSize`	Number of characters in each row	Double
`NFields`	Number of fields in each row	Double array
`FieldFormat`	A 1-by-`NFields` cell containing formats in which each field is encoded. The formats are FORTRAN-77 format codes.	Cell array of strings
`FieldPrecision`	A 1-by-`NFields` cell containing precision of the data in each field	Cell array of strings
`FieldWidth`	A 1-by-`NFields` array containing the number of characters in each field	Double array
`FieldPos`	A 1-by-`NFields` array of numbers representing the starting column for each field	Double array
`DataSize`	Size of the data in the table in bytes	Double
`MissingDataValue`	A 1-by-`NFields` array of numbers used to represent undefined data in each field	Cell array of strings

Field Name	Description	Return Type
Intercept	A 1-by-NFields array of numbers used along with Slope to calculate actual data values from the array data values using the equation: actual_value = Slope*array_value+Intercept	Double array
Slope	A 1-by-NFields array of numbers used with Intercept to calculate true data values from the array data values using the equation: actual_value = Slope*array_value+Intercept	Double array
Offset	Number of bytes from beginning of the file to the location of the first data value in the table	Double
Keywords	A number-of-keywords-by-3 cell array containing all the Keywords, Values and Comments in the ASCII table header	Cell array of strings

BinaryTable Extension

The BinaryTable structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
Rows	Number of rows in the table	Double
RowSize	Number of bytes in each row	Double
NFields	Number of fields in each row	Double

Field Name	Description	Return Type
FieldFormat	A 1-by-NFields cell array containing the data type of the data in each field. The data type is represented by a FITS binary table format code.	Cell array of strings
FieldPrecision	A 1-by-NFields cell containing precision of the data in each field	Cell array of strings
FieldSize	A 1-by-NFields array, where each element contains the number of values in the Nth field	Double array
DataSize	Size of the data in the Binary Table, in bytes. Includes any data past the main table.	Double
MissingDataValue	An 1-by-NFields array of numbers used to represent undefined data in each field	Cell array of double
Intercept	A 1-by-NFields array of numbers used along with Slope to calculate actual data values from the array data values using the equation: actual_value = slope*array_value+Intercept	Double array
Slope	A 1-by-NFields array of numbers used with Intercept to calculate true data values from the array data values using the equation: actual_value = Slope*array_value+Intercept	Double array

Field Name	Description	Return Type
`Offset`	Number of bytes from beginning of the file to the location of the first data value	Double
`ExtensionSize`	Size of any data past the main table, in bytes	Double
`ExtensionOffset`	Number of bytes from the beginning of the file to any data past the main table	Double
`Keywords`	A number-of-keywords-by-3 cell array containing all the Keywords, values, and comments in the Binary Table header	Cell array of strings

Image Extension

The `Image` structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
`DataType`	Precision of the data	String
`Size`	Array containing sizes of each dimension	Double array
`DataSize`	Size of the data in the Image extension in bytes	Double
`Offset`	Number of bytes from the beginning of the file to the first data value	Double
`MissingDataValue`	Value used to represent undefined data	Double

Field Name	Description	Return Type
`Intercept`	Value, used with `Slope`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value+Intercept`	Double
`Slope`	Value, used with `Intercept`, to calculate actual pixel values from the array pixel values, using the equation: `actual_value = Slope*array_value + Intercept`	Double
`Keywords`	A number-of-keywords-by-3 cell array containing all the Keywords, values, and comments in the Binary Table header	Cell array of strings

Unknown Structure

The `Unknown` structure contains the following fields, listed in the order they appear in the structure.

Field Name	Description	Return Type
`DataType`	Precision of the data	String
`Size`	Sizes of each dimension	Double array
`DataSize`	Size of the data in nonstandard extensions, in bytes	Double
`Offset`	Number of bytes from beginning of the file to the first data value	Double

Field Name	Description	Return Type
`MissingDataValue`	Representation of undefined data	Double
`Intercept`	Value, used with `Slope`, to calculate actual data values from the array data values, using the equation: `actual_value = Slope*array_value+Intercept`	Double
`Slope`	Value, used with `Intercept`, to calculate actual data values from the array data values, using the equation: `actual_value = Slope*array_value+Intercept`	Double
`Keywords`	A number-of-keywords-by-3 cell array containing all the Keywords, values, and comments in the Binary Table header	Cell array of strings

Examples

Use `fitsinfo` to obtain information about the FITS file `tst0012.fits`. In addition to its primary data, the file also contains an example of the extensions `BinaryTable`, `Unknown`, `Image`, and `AsciiTable`.

```
S = fitsinfo('tst0012.fits');
S =
         Filename: [1x71 char]
      FileModDate: '12-Mar-2001 18:37:46'
         FileSize: 109440
         Contents: {'Primary' 'Binary Table' 'Unknown'
'Image' 'ASCII Table'}
      PrimaryData: [1x1 struct]
      BinaryTable: [1x1 struct]
```

```
         Unknown: [1x1 struct]
           Image: [1x1 struct]
      AsciiTable: [1x1 struct]
```

The `PrimaryData` field describes the data in the file. For example, the `Size` field indicates the data is a 102-by-109 matrix.

```
S.PrimaryData
          DataType: 'single'
              Size: [102 109]
          DataSize: 44472
  MissingDataValue: []
         Intercept: 0
             Slope: 1
            Offset: 2880
          Keywords: {25x3 cell}
```

The `AsciiTable` field describes the `AsciiTable` extension. For example, using the `FieldWidth` and `FieldPos` fields you can determine the length and location of each field within a row.

```
S.AsciiTable
ans =
              Rows: 53
           RowSize: 59
           NFields: 8
       FieldFormat: {'A9' 'F6.2' 'I3' 'E10.4' 'D20.15' 'A5' 'A1' 'I4'}
    FieldPrecision: {1x8 cell}
        FieldWidth: [9 6.2000 3 10.4000 20.1500 5 1 4]
          FieldPos: [1 11 18 22 33 54 54 55]
          DataSize: 3127
  MissingDataValue: {'*' '---.--' '  *' [] '*' '*' '*' ''}
         Intercept: [0 0 -70.2000 0 0 0 0 0]
             Slope: [1 1 2.1000 1 1 1 1 1]
            Offset: 103680
          Keywords: {65x3 cell}
```

See Also

`fitsread` | `fitswrite` | `fitsdisp` | matlab.io.fits

fitsread

Purpose

Read data from FITS file

Syntax

```
data = fitsread(filename)
data = fitsread(filename,extname)
data = fitsread(filename,extname,index)
data = fitsread(filename,Name,Value)
```

Description

`data = fitsread(filename)` reads the primary data of the Flexible Image Transport System (FITS) file specified by the text string `filename`. The function replaces undefined data values with `NaN` and scales numeric data by the slope and intercept values, always returning double precision values.

`data = fitsread(filename,extname)` reads data from the FITS file extension specified by `extname`.

`data = fitsread(filename,extname,index)` reads data from the FITS file extension specified by `extname` . If there is more than one of the specified extensions in the file, `index` specifies the one to read.

`data = fitsread(filename,Name,Value)` reads data from the FITS file with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

filename

Text string specifying the name of a FITS file.

extname

One of the following text strings specifying the name of a data array or extension in the FITS file. To determine the contents of a FITS, view the `Contents` field of the structure returned by `fitsinfo`.

Data Arrays or Extensions

Extname	Description
'primary'	Read data from the primary data array.
'asciitable'	Read data from the ASCII Table extension. The return value, data, is a 1-D cell array.
'binarytable'	Read data from the Binary Table extension. The return value, data, is a 1-D cell array.
'image'	Read data from the Image extension.
'unknown'	Read data from the Unknown extension.

index

Numeric value specifying which extension to read, if more than one exists in the file.

Name-Value Pair Arguments

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

'info'

`info` structure returned by `fitsinfo` specifying the location of data to read.

Note Using the `info` structure returned by `fitsinfo` to specify the location of data in a FITS file can significantly improve performance, especially when reading multiple images from the file.

'PixelRegion'

Cell array {rows,cols,...} specifying the boundaries of a subimage region to read from the file. Each dimension (rows, cols) is a vector of 1-based indices given either as START, [START STOP], or [START INCREMENT STOP]. This parameter is valid only for primary or image extensions.

'raw'

Specifies that fitsread should not scale the data read from the file or replace undefined values with NaN. Data read from the file is the same class as it is stored in the file.

'TableColumns'

Vector of 1-based indices specifying the columns to read from the ASCII or Binary table extension. This vector should contain unique and valid indices into the table data specified in increasing order. This parameter is valid only for ASCII or Binary extensions.

'TableRows'

Vector of 1-based indices specifying the rows to read from the ASCII or Binary table extension. This vector should contain unique and valid indices into the table data specified in increasing order. This parameter is valid only for ASCII or Binary extensions.

Output Arguments

data

Data returned from the FITS file.

Definitions

extension

A FITS file contains primary data and can optionally contain any number of optional components, called *extensions* in FITS terminology.

Examples

Read primary data from FITS file

```
data = fitsread('tst0012.fits');
```

```
Name          Size              Bytes  Class     Attributes

data        109x102             88944  double
```

Inspect available extensions, read ’image’ extension using the `extname` option.

```
info = fitsinfo('tst0012.fits');
 % List of contents, includes any extensions if present.
 disp(info.Contents);
 imageData = fitsread('tst0012.fits','image');
```

Subsample the fifth plane of ’image’ extension by 2.

```
info        = fitsinfo('tst0012.fits');
 rowend      = info.Image.Size(1);
 colend      = info.Image.Size(2);
 primaryData = fitsread('tst0012.fits','image',...
                      'Info', info,...
                      'PixelRegion',{[1 2 rowend], [1 2 colend], 5 });
```

Read every other row from an ASCII table.

```
info      = fitsinfo('tst0012.fits');
rowend    = info.AsciiTable.Rows;
tableData = fitsread('tst0012.fits','asciitable',...
                    'Info',info,...
                    'TableRows',[1:2:rowend]);
```

Read all data for the first, second and fifth columns of the Binary table.

```
info      = fitsinfo('tst0012.fits');
```

```
rowend    = info.BinaryTable.Rows;
tableData = fitsread('tst0012.fits','binarytable',...
                     'Info',info,...
                     'TableColumns',[1 2 5]);
```

See Also

`fitswrite` | `fitsinfo` | `fitsdisp` | matlab.io.fits

Tutorials

- “Importing Flexible Image Transport System (FITS) Files”

Purpose

Write image to FITS file

Syntax

```
fitswrite(imagedata,filename)
fitswrite(imagedata,filename,Name,Value)
```

Description

`fitswrite(imagedata,filename)` writes `imagedata` to the FITS file specified by `filename`. If `filename` does not exist, `fitswrite` creates the file as a simple FITS file. If `filename` exists, `fitswrite` overwrites the file or appends the image to the end of the file, depending on the value of the `writemode` argument.

`fitswrite(imagedata,filename,Name,Value)` writes `imagedata` to the FITS file specified by `filename` with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

imagedata

Image array.

filename

Text string specifying the name of an existing FITS file or the name you want to assign to a new FITS file.

Name-Value Pair Arguments

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

'WriteMode''

One of these strings:

- `overwrite`
- `append`

Default: `overwrite`

'Compression''

One of these strings:

- `none`
- `gzip`
- `rice`
- `hcompress`
- `plio`

Default: `none`

Examples

Create a FITS file containing the red channel of an RGB image.

```
X = imread('ngc6543a.jpg');
R = X(:,:,1);
fitswrite(R,'myfile.fits');
fitsdisp('myfile.fits');
```

Create a FITS file with three images constructed from the channels of an RGB image.

```
X = imread('ngc6543a.jpg');
R = X(:,:,1);  G = X(:,:,2);  B = X(:,:,3);
fitswrite(R,'myfile.fits');
fitswrite(G,'myfile.fits','writemode','append');
fitswrite(B,'myfile.fits','writemode','append');
fitsdisp('myfile.fits');
```

References

For copyright information, see the `cfitsiocopyright.txt` file.

See Also

`fitsinfo` | `fitsread`matlab.io.fits |

Purpose Round toward zero

Syntax

```
B = fix(A)
```

Description

`B = fix(A)` rounds the elements of `A` toward zero, resulting in an array of integers. For complex `A`, the imaginary and real parts are rounded independently.

Examples

```
a = [-1.9, -0.2, 3.4, 5.6, 7.0, 2.4+3.6i]

a =
  Columns 1 through 4
  -1.9000        -0.2000        3.4000        5.6000

  Columns 5 through 6
   7.0000        2.4000 + 3.6000i

fix(a)

ans =
  Columns 1 through 4
  -1.0000        0             3.0000        5.0000

  Columns 5 through 6
   7.0000        2.0000 + 3.0000i
```

See Also

`ceil` | `floor` | `round`

flintmax

Purpose

Largest consecutive integer in floating-point format

Syntax

```
f = flintmax
f = flintmax(precision)
```

Description

`f = flintmax` returns the largest consecutive integer in IEEE® double precision, which is `2^53`. Above this value, double-precision format does not have integer precision, and not all integers can be represented exactly.

`f = flintmax(precision)` returns the largest consecutive integer in IEEE single or double precision. `flintmax` returns `single(2^24)` for single precision and `2^53` for double precision.

Input Arguments

precision - Floating-point precision type

`'double'` (default) | `'single'`

Floating-point precision type, specified as `'double'` or `'single'`.

Data Types
`char`

Output Arguments

f - Largest consecutive integer in floating-point format

scalar constant

Largest consecutive integer in floating-point format returned as a scalar constant. This constant is `2^53` for double precision and `single(2^24)` for single precision.

Examples

Double Precision

Return the largest consecutive integer in IEEE double precision.

```
format long e
f = flintmax
```

```
f =
```

```
    9.007199254740992e+15
```

This is `2^53`.

Single Precision

Return the largest consecutive integer in IEEE single precision.

```
f = flintmax('single')

f =

    16777216

class(f)

ans =

single
```

This is `single(2^24)`.

Limit of Integer Single Precision

Above the value returned by `flintmax('single')`, not all integers can be represented exactly with single precision.

Return the largest consecutive integer in IEEE single precision.

```
f = flintmax('single')

f =

    16777216
```

This is `single(2^24).`

Add `1` to the value returned from `flintmax`.

```
f1 = f + 1
```

```
f1 =

    16777216
```

f1 is the same as f.

```
isequal(f,f1)
```

```
ans =

     1
```

Add 2 to the value returned from flintmax.

```
f2 = f + 2
```

```
f2 =

    16777218
```

16777218 is represented exactly in single precision while 16777217 is not.

See Also

eps | realmax | intmax | format

Concepts

- “Floating-Point Numbers”

Purpose

Flip array along specified dimension

Syntax

```
B = flipdim(A,dim)
```

Description

`B = flipdim(A,dim)` returns `A` with dimension `dim` flipped.

When the value of `dim` is `1`, the array is flipped row-wise down. When `dim` is `2`, the array is flipped columnwise left to right. `flipdim(A,1)` is the same as `flipud(A)`, and `flipdim(A,2)` is the same as `fliplr(A)`.

Examples

`flipdim(A,1)` where

```
A =

     1     4
     2     5
     3     6
```

produces

```
     3     6
     2     5
     1     4
```

See Also

`fliplr` | `flipud` | `permute` | `rot90`

fliplr

Purpose

Flip matrix left to right

Syntax

```
B = fliplr(A)
```

Description

`B = fliplr(A)` returns `A` with columns flipped in the left-right direction, that is, about a vertical axis.

If `A` is a row vector, then `fliplr(A)` returns a vector of the same length with the order of its elements reversed. If `A` is a column vector, then `fliplr(A)` simply returns `A`.

Examples

If `A` is the 3-by-2 matrix,

```
A =
     1     4
     2     5
     3     6
```

then `fliplr(A)` produces

```
     4     1
     5     2
     6     3
```

If `A` is a row vector,

```
A =
     1     3     5     7     9
```

then `fliplr(A)` produces

```
     9     7     5     3     1
```

Limitations

The array being operated on cannot have more than two dimensions. This limitation exists because the axis upon which to flip a multidimensional array would be undefined.

See Also

`flipdim` | `flipud` | `rot90`

Purpose

Flip matrix up to down

Syntax

```
B = flipud(A)
```

Description

`B = flipud(A)` returns `A` with rows flipped in the up-down direction, that is, about a horizontal axis.

If `A` is a column vector, then `flipud(A)` returns a vector of the same length with the order of its elements reversed. If `A` is a row vector, then `flipud(A)` simply returns `A`.

Examples

If `A` is the 3-by-2 matrix,

```
A =
     1     4
     2     5
     3     6
```

`then flipud(A)` produces

```
     3     6
     2     5
     1     4
```

If `A` is a column vector,

```
A =
     3
     5
     7
```

then `flipud(A)` produces

```
A =
     7
     5
     3
```

Limitations

The array being operated on cannot have more than two dimensions. This limitation exists because the axis upon which to flip a multidimensional array would be undefined.

See Also

`flipdim` | `fliplr` | `rot90`

Purpose

Round toward negative infinity

Syntax

`B = floor(A)`

Description

`B = floor(A)` rounds the elements of `A` to the nearest integers less than or equal to `A`. For complex `A`, the imaginary and real parts are rounded independently.

Examples

```
a = [-1.9  -0.2  3.4  5.6  7.0  2.4+3.6i];

floor(a)

ans =
  -2.0000    -1.0000     3.0000     5.0000     7.0000     2.0000 + 3.0000i
```

See Also

`ceil` | `fix` | `round`

flow

Purpose

Simple function of three variables

Syntax

```
v = flow
v = flow(n)
v = flow(x,y,z)
[x,y,z,v] = flow(...)
```

Description

`flow`, a function of three variables, generates fluid-flow data that is useful for demonstrating `slice`, `interp3`, and other functions that visualize scalar volume data.

`v = flow` produces a 50-by-25-by-25 array.

`v = flow(n)` produces a `n`-by-`2n`-by-`n` array.

`v = flow(x,y,z)` evaluates the speed profile at the points `x`, `y`, and `z`.

`[x,y,z,v] = flow(...)` returns the coordinates as well as the volume data.

See Also

`slice` | `interp3`

How To

- "Slicing Fluid Flow Data"

Purpose

Find minimum of single-variable function on fixed interval

Syntax

```
x = fminbnd(fun,x1,x2)
x = fminbnd(fun,x1,x2,options)
[x,fval] = fminbnd(...)
[x,fval,exitflag] = fminbnd(...)
[x,fval,exitflag,output] = fminbnd(...)
```

Description

`fminbnd` finds the minimum of a function of one variable within a fixed interval.

`x = fminbnd(fun,x1,x2)` returns a value `x` that is a local minimizer of the function that is described in `fun` in the interval `x1 < x < x2`. `fun` is a `function_handle`.

"Parameterizing Functions" in the MATLAB Mathematics documentation, explains how to pass additional parameters to your objective function `fun`.

`x = fminbnd(fun,x1,x2,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `fminbnd` uses these `options` structure fields:

`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output; `'notify'` (default) displays output only if the function does not converge. See "Iterative Display" in MATLAB Mathematics for more information.
`FunValCheck`	Check whether objective function values are valid. `'on'` displays an error when the objective function returns a value that is `complex` or `NaN`. `'off'` displays no error.
`MaxFunEvals`	Maximum number of function evaluations allowed.
`MaxIter`	Maximum number of iterations allowed.

`OutputFcn`	User-defined function that is called at each iteration. See "Output Functions" in MATLAB Mathematics for more information.
`PlotFcns`	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[]`). • `@optimplotx` plots the current point • `@optimplotfval` plots the function value • `@optimplotfunccount` plots the function count See "Plot Functions" in MATLAB Mathematics for more information.
`TolX`	Termination tolerance on `x`.

`[x,fval] = fminbnd(...)` returns the value of the objective function computed in `fun` at `x`.

`[x,fval,exitflag] = fminbnd(...)` returns a value `exitflag` that describes the exit condition of `fminbnd`:

`1`	`fminbnd` converged to a solution `x` based on `options.TolX`.
`0`	Maximum number of function evaluations or iterations was reached.
`-1`	Algorithm was terminated by the output function.
`-2`	Bounds are inconsistent (`x1 > x2`).

`[x,fval,exitflag,output] = fminbnd(...)` returns a structure `output` that contains information about the optimization in the following fields:

`algorithm`	Algorithm used
`funcCount`	Number of function evaluations
`iterations`	Number of iterations
`message`	Exit message

Arguments

`fun` is the function to be minimized. `fun` accepts a scalar `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for a function file

```
x = fminbnd(@myfun,x1,x2);
```

where `myfun.m` is a function file such as

```
function f = myfun(x)
f = ...          % Compute function value at x.
```

or as a function handle for an anonymous function:

```
x = fminbnd(@(x) sin(x*x),x1,x2);
```

Other arguments are described in the syntax descriptions above.

Examples

`x = fminbnd(@cos,3,4)` computes π to a few decimal places and gives a message on termination.

```
[x,fval,exitflag] = ...
   fminbnd(@cos,3,4,optimset('TolX',1e-12,'Display','off'))
```

computes π to about 12 decimal places, suppresses output, returns the function value at `x`, and returns an `exitflag` of 1.

The argument `fun` can also be a function handle for an anonymous function. For example, to find the minimum of the function $f(x) = x^3 - 2x - 5$ on the interval `(0,2)`, create an anonymous function `f`

```
f = @(x)x.^3-2*x-5;
```

Then invoke `fminbnd` with

```
x = fminbnd(f, 0, 2)
```

The result is

```
x =
    0.8165
```

The value of the function at the minimum is

```
y = f(x)

y =
    -6.0887
```

If `fun` is parameterized, you can use anonymous functions to capture the problem-dependent parameters. For example, suppose you want to minimize the objective function `myfun` defined by the following function file:

```
function f = myfun(x,a)
f = (x - a)^2;
```

Note that `myfun` has an extra parameter `a`, so you cannot pass it directly to `fminbind`. To optimize for a specific value of `a`, such as `a = 1.5`.

1. Assign the value to `a`.

   ```
   a = 1.5; % define parameter first
   ```

2. Call `fminbnd` with a one-argument anonymous function that captures that value of `a` and calls `myfun` with two arguments:

   ```
   x = fminbnd(@(x) myfun(x,a),0,1)
   ```

Algorithms

`fminbnd` is a function file. Its algorithm is based on golden section search and parabolic interpolation. Unless the left endpoint x_1 is very

close to the right endpoint x_2, `fminbnd` never evaluates `fun` at the endpoints, so `fun` need only be defined for x in the interval $x_1 < x < x_2$.

If the minimum actually occurs at x_1 or x_2, `fminbnd` returns a point `x` in the interior of the interval (x_1,x_2) that is close to the minimizer. In this case, the distance of `x` from the minimizer is no more than `2*(TolX + 3*abs(x)*sqrt(eps))`. See [1] or [2] for details about the algorithm.

Limitations

The function to be minimized must be continuous. `fminbnd` may only give local solutions.

`fminbnd` often exhibits slow convergence when the solution is on a boundary of the interval.

`fminbnd` only handles real variables.

References

[1] Forsythe, G. E., M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, 1976.

[2] Brent, Richard. P., *Algorithms for Minimization without Derivatives*, Prentice-Hall, Englewood Cliffs, New Jersey, 1973

See Also

`fminsearch` | `fzero` | `optimset` | `function_handle`

How To

- anonymous function

Purpose

Find minimum of unconstrained multivariable function using derivative-free method

Syntax

```
x = fminsearch(fun,x0)
x = fminsearch(fun,x0,options)
[x,fval] = fminsearch(...)
[x,fval,exitflag] = fminsearch(...)
[x,fval,exitflag,output] = fminsearch(...)
```

Description

`fminsearch` finds the minimum of a scalar function of several variables, starting at an initial estimate. This is generally referred to as *unconstrained nonlinear optimization*.

`x = fminsearch(fun,x0)` starts at the point `x0` and returns a value `x` that is a local minimizer of the function described in `fun`. `x0` can be a scalar, vector, or matrix. `fun` is a `function_handle`.

"Parameterizing Functions" in the MATLAB Mathematics documentation explains how to pass additional parameters to your objective function `fun`. See also "Example 2" on page 1-1791 and "Example 3" on page 1-1791 below.

`x = fminsearch(fun,x0,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `fminsearch` uses these `options` structure fields:

`Display`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration; `'final'` displays just the final output; `'notify'` (default) displays output only if the function does not converge. See "Iterative Display" in MATLAB Mathematics for more information.
`FunValCheck`	Check whether objective function values are valid. `'on'` displays an error when the objective function returns a value that is `complex`, `Inf` or `NaN`. `'off'` (the default) displays no error.

MaxFunEvals	Maximum number of function evaluations allowed
MaxIter	Maximum number of iterations allowed
OutputFcn	User-defined function that is called at each iteration. See "Output Functions" in MATLAB Mathematics for more information.
PlotFcns	Plots various measures of progress while the algorithm executes, select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none ([]). • @optimplotx plots the current point • @optimplotfval plots the function value • @optimplotfunccount plots the function count See "Plot Functions" in MATLAB Mathematics for more information.
TolFun	Termination tolerance on the function value
TolX	Termination tolerance on x

[x,fval] = fminsearch(...) returns in fval the value of the objective function fun at the solution x.

[x,fval,exitflag] = fminsearch(...) returns a value exitflag that describes the exit condition of fminsearch:

1	fminsearch converged to a solution x.
0	Maximum number of function evaluations or iterations was reached.
-1	Algorithm was terminated by the output function.

[x,fval,exitflag,output] = fminsearch(...) returns a structure output that contains information about the optimization in the following fields:

`algorithm`	`'Nelder-Mead simplex direct search'`
`funcCount`	Number of function evaluations
`iterations`	Number of iterations
`message`	Exit message

Arguments

`fun` is the function to be minimized. It accepts an input `x` and returns a scalar `f`, the objective function evaluated at `x`. The function `fun` can be specified as a function handle for a function file

```
x = fminsearch(@myfun, x0)
```

where `myfun` is a function file such as

```
function f = myfun(x)
f = ...             % Compute function value at x
```

or as a function handle for an anonymous function, such as

```
x = fminsearch(@(x)sin(x^2), x0);
```

Other arguments are described in the syntax descriptions above.

Examples

Example 1

The Rosenbrock banana function is a classic test example for multidimensional minimization:

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + \left(1 - x_1\right)^2.$$

The minimum is at `(1,1)` and has the value `0`. The traditional starting point is `(-1.2,1)`. The anonymous function shown here defines the function and returns a function handle called `banana`:

```
banana = @(x)100*(x(2)-x(1)^2)^2+(1-x(1))^2;
```

Pass the function handle to `fminsearch`:

```
[x,fval] = fminsearch(banana,[-1.2, 1])
```

This produces

```
x =

    1.0000    1.0000

fval =

    8.1777e-010
```

This indicates that the minimizer was found to at least four decimal places with a value near zero.

Example 2

If `fun` is parameterized, you can use anonymous functions to capture the problem-dependent parameters. For example, suppose you want to minimize the objective function `myfun` defined by the following function file:

```
function f = myfun(x,a)
f = x(1)^2 + a*x(2)^2;
```

Note that `myfun` has an extra parameter `a`, so you cannot pass it directly to `fminsearch`. To optimize for a specific value of `a`, such as `a = 1.5`.

1. Assign the value to `a`.

   ```
   a = 1.5; % define parameter first
   ```

2. Call `fminsearch` with a one-argument anonymous function that captures that value of `a` and calls `myfun` with two arguments:

   ```
   x = fminsearch(@(x) myfun(x,a),[0,1])
   ```

Example 3

You can modify the first example by adding a parameter a to the second term of the banana function:

$$f(x) = 100\left(x_2 - x_1^2\right)^2 + \left(a - x_1\right)^2.$$

This changes the location of the minimum to the point [a,a^2]. To minimize this function for a specific value of a, for example a = sqrt(2), create a one-argument anonymous function that captures the value of a.

```
a = sqrt(2);
banana = @(x)100*(x(2)-x(1)^2)^2+(a-x(1))^2;
```

Then the statement

```
[x,fval] = fminsearch(banana, [-1.2, 1], ...
   optimset('TolX',1e-8));
```

seeks the minimum [sqrt(2), 2] to an accuracy higher than the default on x.

Algorithms

fminsearch uses the simplex search method of Lagarias et al. [1]. This is a direct search method that does not use numerical or analytic gradients.

If n is the length of x, a simplex in n-dimensional space is characterized by the n+1 distinct vectors that are its vertices. In two-space, a simplex is a triangle; in three-space, it is a pyramid. At each step of the search, a new point in or near the current simplex is generated. The function value at the new point is compared with the function's values at the vertices of the simplex and, usually, one of the vertices is replaced by the new point, giving a new simplex. This step is repeated until the diameter of the simplex is less than the specified tolerance.

For more information, see "fminsearch Algorithm".

Limitations

fminsearch can often handle discontinuity, particularly if it does not occur near the solution. fminsearch may only give local solutions.

fminsearch only minimizes over the real numbers, that is, x must only consist of real numbers and $f(x)$ must only return real numbers. When x has complex variables, they must be split into real and imaginary parts.

References

[1] Lagarias, J.C., J. A. Reeds, M. H. Wright, and P. E. Wright, "Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions," *SIAM Journal of Optimization*, Vol. 9 Number 1, pp. 112-147, 1998.

See Also

`fminbnd` | `optimset` | `function_handle`

How To

- anonymous function

fopen

Purpose

Open file, or obtain information about open files

Syntax

```
fileID = fopen(filename)
fileID = fopen(filename, permission)
fileID = fopen(filename, permission, machineformat)
fileID = fopen(filename, permission,
machineformat, encoding)
[fileID, message] = fopen(filename, ...)
fIDs = fopen('all')
[filename, permission, machineformat,
   encoding] = fopen(fileID)
```

Description

`fileID = fopen(filename)` opens the file `filename` for read access, and returns an integer file identifier.

`fileID = fopen(filename, permission)` opens the file with the specified `permission`.

`fileID = fopen(filename, permission, machineformat)` specifies the order for reading or writing bytes or bits in the file.

`fileID = fopen(filename, permission, machineformat, encoding)` specifies the character encoding scheme associated with the file.

`[fileID, message] = fopen(filename, ...)` opens a file. If the operation fails, `message` is a system-dependent error message. Otherwise, `message` is an empty string.

`fIDs = fopen('all')` returns a row vector containing the file identifiers of all open files.

`[filename, permission, machineformat, encoding] = fopen(fileID)` returns the file name, permission, machine format, and encoding that a previous call to `fopen` used when it opened the specified file. `fopen` does not read information from the file to determine these output values. An invalid `fileID` returns empty strings for all output arguments.

Input Arguments

filename

String in single quotation marks that specifies the name of the file to open. Can include a full or partial path.

On UNIX systems, if *`filename`* begins with `'~/'` or `'~`*`username`*`/'`, the `fopen` function expands the path to the current or specified user's home directory, respectively.

If you open a file with read access and `fopen` cannot find *`filename`* in the current folder, `fopen` searches along the MATLAB search path. Otherwise, `fopen` creates a file in the current directory.

permission

String that describes the type of access for the file: read, write, append, or update. Also specifies whether to open files in binary or text mode.

To open files in binary mode, specify one of the following:

`'r'`	Open file for reading (default).
`'w'`	Open or create new file for writing. Discard existing contents, if any.
`'a'`	Open or create new file for writing. Append data to the end of the file.
`'r+'`	Open file for reading and writing.
`'w+'`	Open or create new file for reading and writing. Discard existing contents, if any.
`'a+'`	Open or create new file for reading and writing. Append data to the end of the file.
`'A'`	Append without automatic flushing. (Used with tape drives.)
`'W'`	Write without automatic flushing. (Used with tape drives.)

To read and write to the same file:

- Open the file in update mode (with a *permission* that includes a plus sign, '+').
- Call fseek or frewind between read and write operations. For example, do not call fread followed by fwrite, or fwrite followed by fread, unless you call fseek or frewind between them.

To open files in text mode, attach the letter 't' to the *permission*, such as 'rt' or 'wt+'. For better performance, do not use text mode. The following applies on Windows systems, in text mode:

- Read operations that encounter a carriage return followed by a newline character ('\r\n') remove the carriage return from the input.
- Write operations insert a carriage return before any newline character in the output.

This additional processing is unnecessary for most cases. All MATLAB import functions, and most text editors (including Microsoft Word and WordPad), recognize both '\r\n' and '\n' as newline sequences. However, when you create files for use in Microsoft Notepad, end each line with '\r\n'. For an example, see fprintf.

machineformat

String that specifies the order for reading or writing bytes or bits in the file.

Possible values are:

'n' or 'native'	The byte ordering that your system uses (default)
'b' or 'ieee-be'	Big-endian ordering
'l' or 'ieee-le'	Little-endian ordering

`'s'` or `'ieee-be.l64'`	Big-endian ordering, 64-bit long data type
`'a'` or `'ieee-le.l64'`	Little-endian ordering, 64-bit long data type

By default, all currently supported platforms use little-endian ordering for new files. Existing binary files can use either big- or little-endian ordering.

encoding

String that specifies the character encoding scheme to use for subsequent read and write operations, including `fscanf`, `fprintf`, `fgetl`, `fgets`, `fread`, and `fwrite`.

Supported values are:

`'Big5'`	`'ISO-8859-1'`	`'windows-932'`
`'EUC-JP'`	`'ISO-8859-2'`	`'windows-936'`
`'GBK'`	`'ISO-8859-3'`	`'windows-949'`
`'Macintosh'`	`'ISO-8859-4'`	`'windows-950'`
`'Shift_JIS'`	`'ISO-8859-9'`	`'windows-1250'`
`'US-ASCII'`	`'ISO-8859-13'`	`'windows-1251'`
`'UTF-8'`	`'ISO-8859-15'`	`'windows-1252'`
		`'windows-1253'`
		`'windows-1254'`
		`'windows-1257'`

For a list of additional encoding character sets, see http://www.iana.org/assignments/character-sets. If you specify a value for encoding that is not in the list of supported values, MATLAB issues a warning. Specifying other encodings sometimes (but not always) produces correct results.

Default: system-dependent

Output Arguments

fileID

An integer that identifies the file for all subsequent low-level file I/O operations. If `fopen` cannot open the file, *`fileID`* is `-1`.

MATLAB reserves file identifiers `0`, `1`, and `2` for standard input, standard output (the screen), and standard error, respectively. When `fopen` successfully opens a file, it returns a file identifier greater than or equal to `3`.

message

A system-dependent error message when `fopen` cannot open the specified file. Otherwise, an empty string.

fIDs

Row vector containing the identifiers for all open files, except the identifiers reserved for standard input, output, and error. The number of elements in the vector is equal to the number of open files.

filename

Name of the file associated with the specified *`fileID`*.

permission

The *`permission`* that `fopen` assigned to the file specified by *`fileID`*.

machineformat

The value of *`machineformat`* that `fopen` used when it opened the file specified by *`fileID`*.

encoding

The character encoding scheme that `fopen` associated with the file specified by *`fileID`*.

The value that `fopen` returns for *`encoding`* is a standard character encoding scheme name. It is not always the same as the *`encoding`* argument that you used in the call to `fopen` to open the file.

Examples

Open a file. Pass the file identifier, `fid`, to other file I/O functions to read data and close the file.

```
fid = fopen('fgetl.m');

tline = fgetl(fid);
while ischar(tline)
    disp(tline);
    tline = fgetl(fid);
end

fclose(fid);
```

Create a prompt to request the name of a file to open. If `fopen` cannot open the file, display the relevant error message.

```
fid = -1;
msg = '';
while fid < 0
   disp(msg);
   filename = input('Open file: ', 's');
   [fid,msg] = fopen(filename);
end
```

Open a file to write Unicode® characters to a file using the Shift-JIS character encoding scheme:

```
fid = fopen('japanese_out.txt', 'w', 'n', 'Shift_JIS');
```

See Also

`fclose` | `ferror` | `fseek` | `ftell` | `feof` | `fscanf` | `fprintf` | `fread` | `fwrite`

fopen (serial)

Purpose

Connect serial port object to device

Syntax

```
fopen(obj)
```

Description

`fopen(obj)` connects the serial port object, `obj` to the device.

Tips

Before you can perform a read or write operation, `obj` must be connected to the device with the `fopen` function. When `obj` is connected to the device:

- Data remaining in the input buffer or the output buffer is flushed.
- The `Status` property is set to `open`.
- The `BytesAvailable`, `ValuesReceived`, `ValuesSent`, and `BytesToOutput` properties are set to 0.

An error is returned if you attempt to perform a read or write operation while `obj` is not connected to the device. You can connect only one serial port object to a given device.

Some properties are read-only while the serial port object is open (connected), and must be configured before using `fopen`. Examples include `InputBufferSize` and `OutputBufferSize`. Refer to the property reference pages to determine which properties have this constraint.

The values for some properties are verified only after `obj` is connected to the device. If any of these properties are incorrectly configured, then an error is returned when `fopen` is issued and `obj` is not connected to the device. Properties of this type include `BaudRate`, and are associated with device settings.

If you use the `help` command to display help for `fopen`, then you need to supply the pathname shown below.

```
help serial/fopen
```

Examples

This example creates the serial port object `s`, connects `s` to the device using `fopen`, writes and reads text data, and then disconnects `s` from the device. This example works on a Windows platform.

```
s = serial('COM1');
fopen(s)
fprintf(s,'*IDN?')
idn = fscanf(s);
fclose(s)
```

See Also

fclose | BytesAvailable | BytesToOutput | Status | ValuesReceived | ValuesSent

Purpose

Execute statements specified number of times

Syntax

```
for index = values
   program statements
          :
end
```

Description

`for index=values, program statements, end` repeatedly executes one or more MATLAB statements in a loop. *`values`* has one of the following forms:

`initval``:`*`endval`*	increments the *`index`* variable from *`initval`* to *`endval`* by `1`, and repeats execution of *`program statements`* until *`index`* is greater than *`endval`*.
`initval``:`*`step`*`:`*`endval`*	increments *`index`* by the value *`step`* on each iteration, or decrements when *`step`* is negative.
`valArray`	creates a column vector *`index`* from subsequent columns of array *`valArray`* on each iteration. For example, on the first iteration, *`index`* `=` *`valArray`*`(:,1)`. The loop executes for a maximum of *`n`* times, where *`n`* is the number of columns of *`valArray`*, given by `numel(`*`valArray`*`, 1, :)`. The input *`valArray`* can be of any MATLAB data type, including a string, cell array, or struct.

Tips

- To force an immediate exit of the loop, use a `break` or `return` statement. To skip the rest of the instructions in the loop, increment the loop counter, and begin the next iteration, use a `continue` statement.
- Avoid assigning a value to the *`index`* variable within the body of a loop. The `for` statement overrides any changes made to the *`index`* within the loop.

- To iterate over the values of a single column vector, first transpose it to create a row vector.

Examples

Create a Hilbert matrix using nested `for` loops:

```
k = 10;
hilbert = zeros(k,k);      % Preallocate matrix

for m = 1:k
    for n = 1:k
        hilbert(m,n) = 1/(m+n -1);
    end
end
```

Step by increments of `-0.1`, and display the step values:

```
for s = 1.0: -0.1: 0.0
   disp(s)
end
```

Execute statements for a defined set of index values:

```
for s = [1,5,8,17]
   disp(s)
end
```

Successively set `e` to unit vectors:

```
for e = eye(5)
  disp('Current value of e:')
  disp(e)
end
```

See Also

`end` | `while` | `break` | `continue` | `parfor` | `return` | `if` | `switch` | `colon`

Purpose

Set display format for output

Syntax

```
format
format Style
```

Description

`format` sets the display of floating-point numeric values to the default display format, which is the short fixed decimal format. This format displays 5-digit scaled, fixed-point values.

The `format` function affects only how numbers display in the Command Window, not how MATLAB computes or saves them.

`format Style` changes the display format to the specified `Style`.

Input Arguments

Style - Output display format

`short` (default) | `long` | `shortE` | `longE`

Output display format, specified as one of the strings listed in the tables that follow.

Use these styles to switch between different output display formats for floating-point variables. Styles are case insensitive. You also can insert a space between `short` or `long` and the presentation type, for instance, `format short E`.

Style	Result	Example
`short` (default)	Short fixed decimal format, with 4 digits after the decimal point. If you are displaying a matrix with a wide range of values, consider using `shortG`. See "Display Large Data Range in short and shortg Formats" on page 1-1810	`3.1416`
`long`	Long fixed decimal format, with 15 digits after the decimal point for `double` values, and 7 digits	`3.141592653589793`

Style	Result	Example
	after the decimal point for `single` values.	
`shortE`	Short scientific notation, with 4 digits after the decimal point. Integer-valued floating-point numbers with a maximum of 9 digits do not display in scientific notation.	`3.1416e+00`
`longE`	Long scientific notation, with 15 digits after the decimal point for `double` values, and 7 digits after the decimal point for `single` values. Integer-valued floating-point numbers with a maximum of 9 digits do not display in scientific notation.	`3.141592653589793e+00`
`shortG`	The more compact of short fixed decimal or scientific notation, with 5 digits.	`3.1416`
`longG`	The more compact of long fixed decimal or scientific notation, with 15 digits for `double` values, and 7 digits for `single` values.	`3.14159265358979`
`shortEng`	Short engineering notation, with 4 digits after the decimal point, and an exponent that is a multiple of 3.	`3.1416e+000`
`longEng`	Long engineering notation, with 15 significant digits, and an exponent that is a multiple of 3.	`3.14159265358979e+000`

Use these format styles to switch between different output display formats for all numeric variables.

Style	Result	Example
`+`	Positive/Negative format, with `+`, `-`, and blank characters displayed for positive, negative, and zero elements.	`+`
`bank`	Currency format, with 2 digits after the decimal point.	`3.14`
`hex`	Hexadecimal representation of a binary double-precision number.	`400921fb54442d18`
`rat`	Ratio of small integers.	`355/113`

Use these format styles to affect the spacing in the display of all variables.

DispType	Result	Example

`compact`	Suppresses excess line feeds to show more output in a single screen. Contrast with `loose`.	`theta = pi/2`
		`theta =`
		`1.5708`

`loose`	Adds linefeeds to make output more readable. Contrast with `compact`.	`theta = pi/2`
		
		`theta =`
		
		`1.5708`

Examples

Change and View Current Format

Set the display format to long fixed decimal.

```
format long
```

View the result for the value of `pi`.

```
pi
```

```
ans =

   3.141592653589793
```

View the current format.

```
get(0,'format')
```

```
ans =

long
```

Display Values in Default and Hexadecimal Format

Set the format to its default, and display the maximum values for integers and real numbers in MATLAB.

```
format

intmax('uint64')
```

```
ans =

 18446744073709551615
```

```
realmax
```

```
ans =
  1.7977e+308
```

Change the display format to hexadecimal, and then display the same values.

```
format hex
```

```
intmax('uint64')

ans =

   ffffffffffffffff

realmax

ans =

   7fefffffffffffff
```

The hexadecimal display corresponds to the internal representation of the value. It is not the same as the hexadecimal notation in the C programming language.

View Output in Short and Long Engineering Notation

View the difference between output displayed in the `shortEng` and `longEng` formats.

Set the display format to `shortEng`.

```
format shortEng
```

Create variable `A` and increase its value by a multiple of 10 each time through a `for` loop. Display each value of `A`.

```
A = 5.123456789;
for k=1:10
   disp(A)
   A = A * 10;
end

     5.1235e+000
    51.2346e+000
   512.3457e+000
     5.1235e+003
    51.2346e+003
```

```
   512.3457e+003
     5.1235e+006
    51.2346e+006
   512.3457e+006
     5.1235e+009
```

The values for A display with 4 digits after the decimal point, and an exponent that is a multiple of 3.

Set the display format to longEng and view the same values of A.

```
format longEng

A = 5.123456789;
for k=1:10
   disp(A)
   A = A * 10;
end
```

```
    5.12345678900000e+000
    51.2345678900000e+000
    512.345678900000e+000
    5.12345678900000e+003
    51.2345678900000e+003
    512.345678900000e+003
    5.12345678900000e+006
    51.2345678900000e+006
    512.345678900000e+006
    5.12345678900000e+009
```

The values for A display with 15 digits, and an exponent that is a multiple of 3.

Display Large Data Range in short and shortg Formats

View the difference between the short and shortg formats when the values in a matrix span a wide range.

Define variable x and display the output in the short format.

```
x = [25 56 255 9876899999];
format short
x

x =
   1.0e+09 *
    0.0000    0.0000    0.0000    9.8769
```

The display indicates each value is multiplied by `1.0e+09`.

Set the format to `shortg` and redisplay `x`.

```
format shortg
x

x =
           25            56           255    9.8769e+09
```

Algorithms

MATLAB always displays integer variables to the appropriate number of digits for the class. For example, MATLAB uses 3 digits to display numbers of type `int8` (for example, -128:127). Setting `format` to `short` or `long` does not affect the display of integer variables.

If the largest element of a matrix is larger than 10^3 or smaller than 10^{-3}, then MATLAB applies a common scale factor for the short and long formats.

Tips

- The specified format applies only to the current MATLAB session. To maintain a format across sessions, choose a **Numeric format** or **Numeric display** option in the Command Window Preferences.
- To see which `Style` is currently in use, type

  ```
  get(0,'Format')
  ```

  To see if `compact` or `loose` formatting is currently selected, type

  ```
  get(0,'FormatSpacing')
  ```

See Also

`disp` | `display` | `isnumeric` | `isfloat` | `isinteger` | `floor` | `sprintf` | `fprintf` | `num2str` | `rat` | `spy`

Concepts

- "Format Output in Command Window"

Purpose

Plot function between specified limits

Syntax

```
fplot(fun,limits)
fplot(fun,limits,LineSpec)
fplot(fun,limits,tol)
fplot(fun,limits,tol,LineSpec)
fplot(fun,limits,n)
fplot(fun,lims,...)
fplot(axes_handle,...)
[X,Y] = fplot(fun,limits,...)
```

Description

`fplot` plots a function between specified limits. The function must be of the form $y = f(x)$, where x is a vector whose range specifies the limits, and `y` is a vector the same size as `x` and contains the function's value at the points in x (see the first example). If the function returns more than one value for a given x, then y is a matrix whose columns contain each component of $f(x)$ (see the second example).

`fplot(fun,limits)` plots `fun` between the limits specified by `limits`. `limits` is a vector specifying the x-axis limits (`[xmin xmax]`), or the x- and y-axes limits, (`[xmin xmax ymin ymax]`).

`fun` must be

- The name of a function
- A string with variable `x` that may be passed to `eval`, such as `'sin(x)'`, `'diric(x,10)'`, or `'[sin(x),cos(x)]'`
- A function handle

The function $f(x)$ must return a row vector for each element of vector x. For example, if $f(x)$ returns `[f1(x),f2(x),f3(x)]` then for input `[x1;x2]` the function should return the matrix

```
f1(x1) f2(x1) f3(x1)
f1(x2) f2(x2) f3(x2)
```

`fplot(fun,limits,LineSpec)` plots `fun` using the line specification `LineSpec`.

`fplot(fun,limits,tol)` plots `fun` using the relative error tolerance `tol` (the default is `2e-3`, i.e., 0.2 percent accuracy).

`fplot(fun,limits,tol,LineSpec)` plots `fun` using the relative error tolerance `tol` and a line specification that determines line type, marker symbol, and color. See `LineSpec` for more information.

`fplot(fun,limits,n)` with `n >= 1` plots the function with a minimum of `n+1` points. The default `n` is `1`. The maximum step size is restricted to be `(1/n)*(xmax-xmin)`.

`fplot(fun,lims,...)` accepts combinations of the optional arguments `tol`, `n`, and `LineSpec`, in any order.

`fplot(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`[X,Y] = fplot(fun,limits,...)` returns the abscissas and ordinates for `fun` in `X` and `Y`. No plot is drawn on the screen; however, you can plot the function using `plot(X,Y)`.

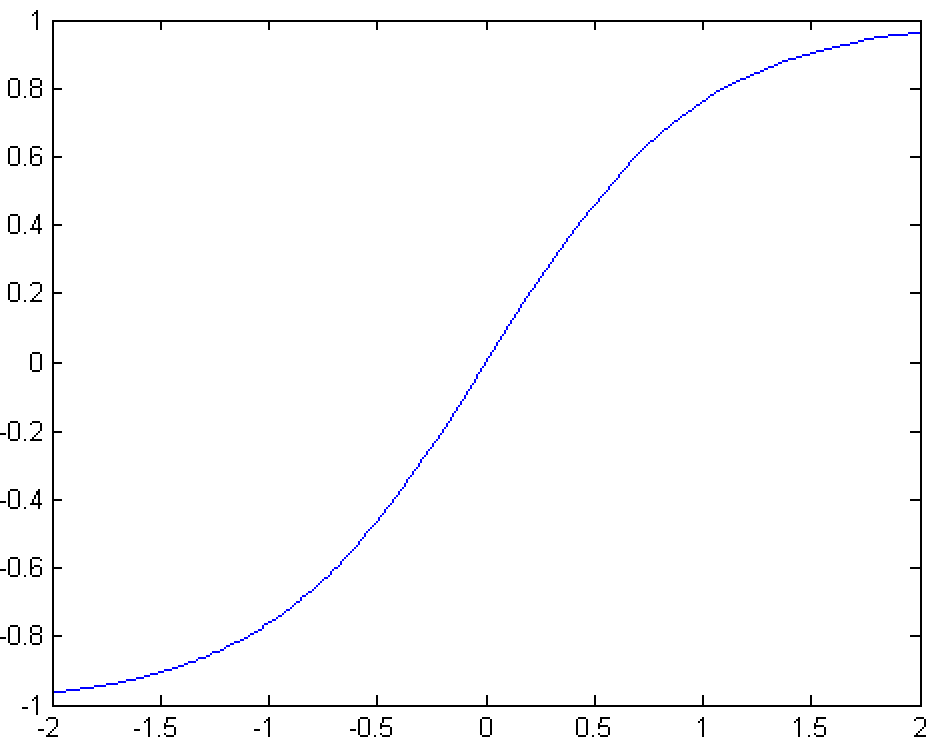
Tips

`fplot` uses adaptive step control to produce a representative graph, concentrating its evaluation in regions where the function's rate of change is the greatest.

Examples

Plot the hyperbolic tangent function from -2 to 2:

```
fnch = @tanh;
fplot(fnch,[-2 2])
```



Create a file, `myfun`, that returns a two-column matrix:

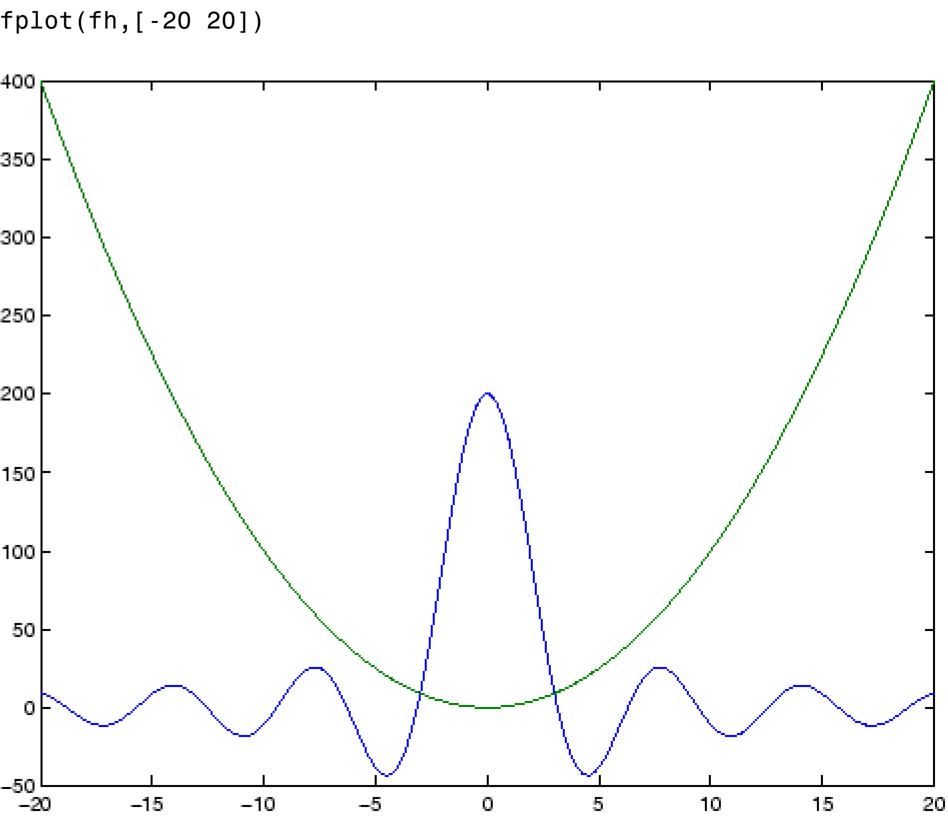
```
function Y = myfun(x)
Y(:,1) = 200*sin(x(:))./x(:);
Y(:,2) = x(:).^2;
```

Create a function handle pointing to `myfun`:

```
fh = @myfun;
```

Plot the function with the statement

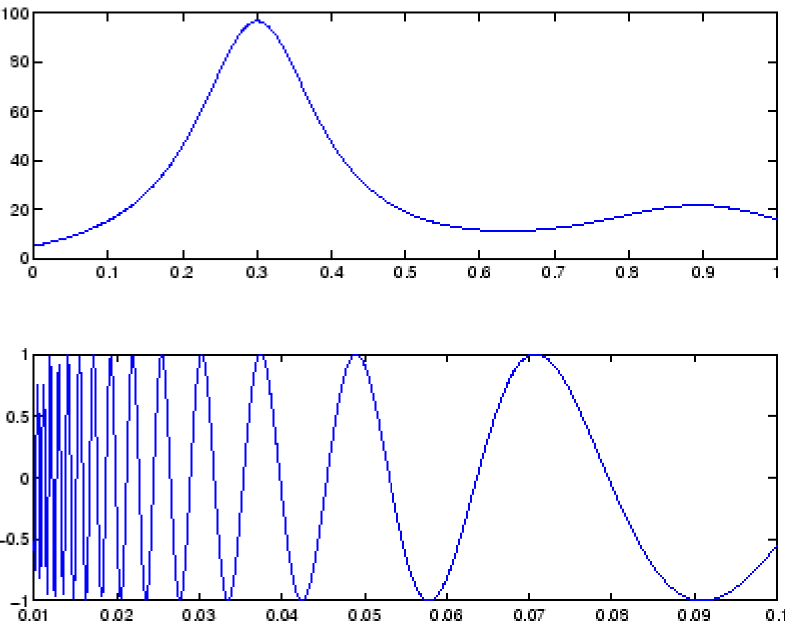
```
fplot(fh,[-20 20])
```



Additional Example

This example passes function handles to `fplot`, one created from a MATLAB function and the other created from an anonymous function.

```
hmp = @humps;
subplot(2,1,1);fplot(hmp,[0 1])
sn = @(x) sin(1./x);
subplot(2,1,2);fplot(sn,[.01 .1])
```



See Also

`eval` | `ezplot` | `feval` | `LineSpec` | `plot`

How To

- Anonymous Functions

fprintf

Purpose

Write data to text file

Syntax

```
fprintf(fileID,formatSpec,A1,...,An)
nbytes = fprintf(fileID,formatSpec,A1,...,An)

fprintf(formatSpec,A1,...,An)
```

Description

`fprintf(fileID,formatSpec,A1,...,An)` applies the `formatSpec` to all elements of arrays `A1,...An` in column order, and writes the data to a text file. `fprintf` uses the encoding scheme specified in the call to `fopen`.

`nbytes = fprintf(fileID,formatSpec,A1,...,An)` additionally returns the number of bytes that `fprintf` writes.

`fprintf(formatSpec,A1,...,An)` formats data and displays the results on the screen.

Input Arguments

fileID - File identifier

`1` (default) | `2` | scalar

File identifier, specified as one of the following:

- A file identifier obtained from `fopen`.
- `1` for standard output (the screen).
- `2` for standard error.

Data Types

`double`

formatSpec - Format of the output fields

string

Format of the output fields, specified as a string.

The string can include a percent sign followed by a conversion character. The following table lists the available conversion characters and subtypes.

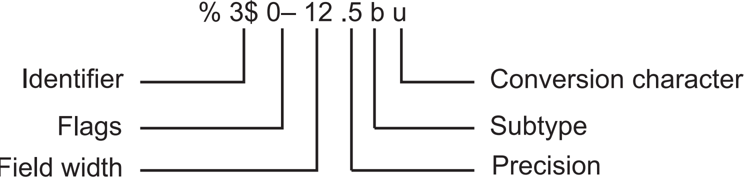
Value Type	Conversion	Details
Integer, signed	`%d` or `%i`	Base 10
Integer, unsigned	`%u`	Base 10
	`%o`	Base 8 (octal)
	`%x`	Base 16 (hexadecimal), lowercase letters `a`–`f`
	`%X`	Same as `%x`, uppercase letters `A`–`F`
Floating-point number	`%f`	Fixed-point notation
	`%e`	Exponential notation, such as `3.141593e+00`
	`%E`	Same as `%e`, but uppercase, such as `3.141593E+00`
	`%g`	The more compact of `%e` or `%f`, with no trailing zeros
	`%G`	The more compact of `%E` or `%f`, with no trailing zeros

	`%bx` or `%bX`
	`%bo`
	`%bu`
		Double-precision hexadecimal, octal, or decimal value
		Example: `%bx` prints `pi` as `400921fb54442d18`

	`%tx` or `%tX`
	`%to`
	`%tu`
		Single-precision hexadecimal, octal, or decimal value
		Example: `%tx` prints `pi` as `40490fdb`

Characters	`%c`	Single character
	`%s`	String of characters

The string can include optional operators, which appear in the following order (includes spaces for clarity):



Optional operators include:

- Identifier

  Order for processing inputs. Use the syntax *n*$, where *n* represents the position of the value in the input list.

  For example, `'%3$s %2$s %1$s %2$s'` prints inputs `'A'`, `'B'`, `'C'` as follows: `C B A B`.

- Flags

`'–'`	Left-justify. Example: `%-5.2f`
`'+'`	Print sign character (+) for positive values. Example: `%+5.2f`
`' '`	Pad to field width with spaces before the value. Example: `% 5.2f`

'0'	Pad to field width with zeros. Example: `%05.2f`
'#'	Modify selected numeric conversions: • For `%o`, `%x`, or `%X`, print `0`, `0x`, or `0X` prefix. • For `%f`, `%e`, or `%E`, print decimal point even when precision is 0. • For `%g` or `%G`, do not remove trailing zeros or decimal point. Example: `%#5.0f`

- Field width

  Minimum number of characters to print. Can be a number, or an asterisk (`*`) to refer to an argument in the input list. For example, the input list `('%12d', intmax)` is equivalent to `('%*d', 12, intmax)`.

- Precision

For `%f`, `%e`, or `%E`:	Number of digits to the right of the decimal point. Example: `'%6.4f'` prints `pi` as `'3.1416'`
For `%g` or `%G`	Number of significant digits. Example: `'%6.4g'` prints `pi` as `' 3.142'`

  Can be a number, or an asterisk (`*`) to refer to an argument in the input list. For example, the input list `('%6.4f', pi)` is equivalent to `('%*.*f', 6, 4, pi)`.

The string can also include combinations of the following:

- Literal text to print. To print a single quotation mark, include `''` in `formatSpec`.
- Control characters, including:

`%%`	Percent character
`\\`	Backslash
`\a`	Alarm
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\xN`	Character whose ASCII code is the hexadecimal number, *N*
`\N`	Character whose ASCII code is the octal number, *N*

The following limitations apply to conversions:

- Numeric conversions print only the real component of complex numbers.
- If you specify a conversion that does not fit the data, such as a string conversion for a numeric value, MATLAB overrides the specified conversion, and uses `%e`.
- If you apply a string conversion (`%s`) to integer values, MATLAB converts values that correspond to valid character codes to characters. For example, `'%s'` converts `[65 66 67]` to `ABC`.

A1,...,An - Numeric or character arrays

scalar | vector | matrix | multidimensional array

Numeric or character arrays, specified as a scalar, vector, matrix, or multidimensional array.

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

Tips

- Format specifiers for the reading functions `sscanf` and `fscanf` differ from the formats for the writing functions `sprintf` and `fprintf`. The reading functions do not support a precision field. The width field specifies a minimum for writing but a maximum for reading.

Examples

Print Literal Text and Array Values

Print multiple numeric values and literal text to the screen.

```
A1 = [9.9, 9900];
A2 = [8.8,  7.7 ; ...
      8800, 7700];
formatSpec = 'X is %4.2f meters or %8.3f mm\n';
fprintf(formatSpec,A1,A2)
```

```
X is 9.90 meters or 9900.000 mm
X is 8.80 meters or 8800.000 mm
X is 7.70 meters or 7700.000 mm
```

Print Double-Precision Values as Integers

Explicitly convert double-precision values with fractions to integer values.

```
a = [1.02, 3.04, 5.06];
fprintf('%d\n',round(a));
```

```
1
3
5
```

Write Table to Text File

Write a short table of the exponential function to a text file called `exp.txt`.

```
x = 0:.1:1;
A = [x; exp(x)];

fileID = fopen('exp.txt','w');
fprintf(fileID,'%6s %12s\n','x','exp(x)');
fprintf(fileID,'%6.2f %12.8f\n',A);
fclose(fileID);
```

The first call to `fprintf` prints header text `x` and `exp(x)`, and the second call prints the values from variable `A`.

If you plan to read the file with Microsoft Notepad, use `'\r\n'` instead of `'\n'` to move to a new line. For example, replace the calls to `fprintf` with the following:

```
fprintf(fileID,'%6s %12s\r\n','x','exp(x)');
fprintf(fileID,'%6.2f %12.8f\r\n',A);
```

MATLAB import functions, all UNIX applications, and Microsoft Word and WordPad recognize `'\n'` as a newline indicator.

View the contents of the file with the `type` command.

```
type exp.txt
```

Display Hyperlinks in Command Window

Display a hyperlink (The MathWorks Web Site) on the screen.

```
site = 'http://www.mathworks.com';
title = 'The MathWorks Web Site';

fprintf('<a href = "%s">%s</a>\n',site,title)
```

References

[1] Kernighan, B. W., and D. M. Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, Inc., 1988.

[2] ANSI specification X3.159-1989: “Programming Language C,” ANSI, 1430 Broadway, New York, NY 10018.

See Also

`disp` | `fclose` | `ferror` | `fopen` | `fread` | `fscanf` | `fwrite` | `fseek` | `ftell` | `sprintf`

Related Examples

- “Exporting a Cell Array to a Text File”
- “Appending or Overwriting Existing Files”

Concepts

- “Formatting Strings”

fprintf (serial)

Purpose

Write text to device

Syntax

```
fprintf(obj,'cmd')
fprintf(obj,'format','cmd')
fprintf(obj,'cmd','mode')
fprintf(obj,'format','cmd','mode')
```

Description

`fprintf(obj,'cmd')` writes the string `cmd` to the device connected to the serial port object, `obj`. The default format is `%s\n`. The write operation is synchronous and blocks the command-line until execution completes.

`fprintf(obj,'format','cmd')` writes the string using the format specified by *`format`*.

`fprintf(obj,'cmd','mode')` writes the string with command line access specified by *`mode`*. *`mode`* specifies if `cmd` is written synchronously or asynchronously.

`fprintf(obj,'format','cmd','mode')` writes the string using the specified format. *`format`* is a C language conversion specification.

You need an open connection from the serial port object, `obj`, to the device before performing read or write operations.

Use the `fopen` function to open a connection to the device. When `obj` has an open connection to the device it has a `Status` property value of `open`. Refer to "Troubleshooting Common Errors" for `fprintf` errors.

To understand the use of `fprintf` refer to " Completing a Write Operation with fprintf" and "Rules for Writing the Terminator".

Input Arguments

format

ANSI C conversion specification includes these conversion characters.

Specifier	Description
`%c`	Single character
`%d` or `%i`	Decimal notation (signed)

Specifier	Description
`%e`	Exponential notation (using lowercase `e` as in `3.1415e+00`)
`%E`	Exponential notation (using uppercase `E` as in `3.1415E+00`)
`%f`	Fixed-point notation
`%g`	The more compact of `%e` or `%f`, as defined above. Insignificant zeros do not print.
`%G`	Same as `%g`, but using uppercase `E`
`%o`	Octal notation (unsigned)
`%s`	String of characters
`%u`	Decimal notation (unsigned)
`%x`	Hexadecimal notation (using lowercase letters `a`–`f`)
`%X`	Hexadecimal notation (using uppercase letters `A`–`F`)

mode

Specifies whether the string `cmd` is written synchronously or asynchronously:

- `sync`: `cmd` is written synchronously and the command line is blocked.
- `async`: `cmd` is written asynchronously and the command line is not blocked.

If *`mode`* is not specified, the write operation is synchronous.

If you specify asynchronous *`mode`*, when the write operation occurs:

- The `BytesToOutput` property value continuously updates to reflect the number of bytes in the output buffer.
- The MATLAB file callback function specified for the `OutputEmptyFcn` property is executed when the output buffer is empty.

Use the `TransferStatus` property to determine whether an asynchronous write operation is in progress.

For more information on synchronous and asynchronous write operations, see Controlling Access to the MATLAB Command Line.

Examples

Create a serial port object `s` and connect it to a Tektronix TDS 210 oscilloscope. Write the `RS232?` command with `fprintf`. `RS232?` instructs the scope to return serial port communications settings. This example works on a Windows platform.

```
s = serial('COM1');
fopen(s)
fprintf(s,'RS232?')
```

Specify a format for the data that does not include the terminator, or configure the terminator to empty.

```
s = serial('COM1');
fopen(s)
fprintf(s,'%s','RS232?')
```

The default format for `fprintf` is `%s\n`. Therefore, the terminator specified by the `Terminator` property is automatically written. However, in some cases you might want to suppress writing the terminator.

Specify an array of formats and commands:

```
s = serial('COM1');
fopen(s)
fprintf(s,['ch:%d scale:%d'],[1 20e-3],'sync');
```

See Also

`fopen` | `fwrite` | `stopasync` | `BytesToOutput` | `OutputBufferSize` | `OutputEmptyFcn` | `Status` | `TransferStatus` | `ValuesSent`

Tutorials

- Writing and Reading Data
- Controlling Access to the MATLAB Command Line

Purpose

Return image data associated with movie frame

Syntax

```
[X,Map] = frame2im(F)
```

Description

`[X,Map] = frame2im(F)` returns the indexed image `X` and associated colormap `Map` from the single movie frame `F`. If the frame contains true-color data, the m-by-n-3 matrix `Map` is empty. The functions `getframe` and `im2frame` create a movie frame.

Examples

Create and capture an image using `getframe` and `frame2im`:

```
peaks                          %Make figure
f = getframe;                  %Capture screen shot
[im,map] = frame2im(f);        %Return associated image data
if isempty(map)                %Truecolor system
  rgb = im;
else                           %Indexed system
  rgb = ind2rgb(im,map);       %Convert image data
end
```

See Also

`getframe` | `im2frame` | `movie`

fread

Purpose

Read data from binary file

Syntax

```
A = fread(fileID)
A = fread(fileID, sizeA)
A = fread(fileID, sizeA, precision)
A = fread(fileID, sizeA, precision, skip)
A = fread(fileID, sizeA, precision, skip, machineformat)
[A, count] = fread( ___ )
```

Description

`A = fread(fileID)` reads data from a binary file into column vector *A* and positions the file pointer at the end-of-file marker.

`A = fread(fileID, sizeA)` reads *sizeA* elements into *A* and positions the file pointer after the last element read. *sizeA* can be an integer, or can have the form `[m,n]`.

`A = fread(fileID, sizeA, precision)` interprets values in the file according to the form and size described by the *precision*. The *sizeA* parameter is optional.

`A = fread(fileID, sizeA, precision, skip)` skips *skip* bytes after reading each value. If *precision* is `bitn` or `ubitn`, specify *skip* in bits. The *sizeA* parameter is optional.

`A = fread(fileID, sizeA, precision, skip, machineformat)` reads data with the specified *machineformat*. The *sizeA* and *skip* parameters are optional.

`[A, count] = fread( ___ )` returns the number of elements `fread` reads into *A*.

Input Arguments

fileID

An integer file identifier obtained from `fopen`.

sizeA

Dimensions of the output array *A*. Specify in one of the following forms:

`inf`	Column vector with the number of elements in the file. (default)
n	Column vector with *n* elements.
[*m*,*n*]	*m*-by-*n* matrix, filled in column order. *n* can be `inf`, but *m* cannot.

precision

String that specifies the form and size of the values to read. Optionally includes the class for the output matrix *A*.

Use one of the following forms:

'*source*'	Specifies class of input values. Output matrix *A* is class `double`. Example: `'int16'`
'*source*=>*output*'	Specifies classes of input and output. Example: `'int8=>char'`
'**source*'	Output has the same class as input. Example: `'*uint8'` For '`bit`*n*' or '`ubit`*n*' precisions, output has the smallest class that can contain the input. Example: `'*ubit18'` is equivalent to `'ubit18=>uint32'`
'*N***source*' OR '*N***source*=>*output*'	Valid if you specify a *skip* parameter. Read *N* values before each skip. Example: `'4*int8'`

The following table shows possible values for *source* and *output*.

Value Type	Precision	Bits (Bytes)
Integers, unsigned	uint	32 (4)
	uint8	8 (1)
	uint16	16 (2)
	uint32	32 (4)
	uint64	64 (8)
	uchar	8 (1)
	unsigned char	8 (1)
	ushort	16 (2)
	ulong	system-dependent
	ubit*n*	$1 \le n \le 64$
Integers, signed	int	32 (4)
	int8	8 (1)
	int16	16 (2)
	int32	32 (4)
	int64	64 (8)
	integer*1	8 (1)
	integer*2	16 (2)
	integer*4	32 (4)
	integer*8	64 (8)
	schar	8 (1)
	signed char	8 (1)
	short	16 (2)
	long	system-dependent
	bit*n*	$1 \le n \le 64$

Value Type	Precision	Bits (Bytes)
Floating-point numbers	`single`	`32 (4)`
	`double`	`64 (8)`
	`float`	`32 (4)`
	`float32`	`32 (4)`
	`float64`	`64 (8)`
	`real*4`	`32 (4)`
	`real*8`	`64 (8)`
Characters	`char*1`	`8 (1)`
	`char`	Depends on the encoding scheme associated with the file. Set encoding with `fopen`.

On 32-bit systems, `long` and `ulong` are 32 bits. On 64-bit systems, the sizes of `long` and `ulong` correspond to the data model that the C compiler uses: 32 bits on Microsoft Windows systems, 64 bits on other 64-bit systems.

For most *`source`* precisions, if `fread` reaches the end of the file before reading a complete element, `fread` does not return a value for the final element. However, if *`source`* is `bit`*`n`* or `ubit`*`n`*, then `fread` returns a partial result for the final element.

Default: `'uint8=>double'`

skip

Number of bytes to skip after reading each value. If you specify a *`precision`* of `bit`*`n`* or `ubit`*`n`*, specify *`skip`* in bits. Use this parameter to read data from noncontiguous fields in fixed-length records.

Default: 0

machineformat

String that specifies the order for reading bytes within the file. For `bitn` and `ubitn` precisions, specifies the order for reading bits within a byte.

Possible values are:

`'n'` or `'native'`	The byte ordering that your system uses (default)
`'b'` or `'ieee-be'`	Big-endian ordering
`'l'` or `'ieee-le'`	Little-endian ordering
`'s'` or `'ieee-be.l64'`	Big-endian ordering, 64-bit long data type
`'a'` or `'ieee-le.l64'`	Little-endian ordering, 64-bit long data type

By default, all currently supported platforms use little-endian ordering for new files. Existing binary files can use either big- or little-endian ordering.

Output Arguments

A

A column vector, unless you specify *sizeA* with the form `[m,n]`. Data in *A* is class `double` unless you specify a different class in the *precision* argument.

count

The number of elements that `fread` successfully reads.

Examples

Read the contents of a file:

```
% Create the file
fid = fopen('magic5.bin', 'w');
fwrite(fid, magic(5));
fclose(fid);

% Read the contents back into an array
```

```
fid = fopen('magic5.bin');
m5 = fread(fid, [5, 5], '*uint8');
fclose(fid);
```

Simulate the `type` function with `fread`, to display the contents of a text file:

```
fid = fopen('fread.m');

% read the entire file as characters
% transpose so that F is a row vector
F = fread(fid, '*char')'

fclose(fid);
```

If you do not specify the precision, `fread` applies the default `uint8=>double`:

```
fid = fopen('fread.m');
F_nums = fread(fid, 6)'  % read the first 6 bytes ('%FREAD')
fclose(fid);
```

This code returns

```
F_nums =
    37    70    82    69    65    68
```

Read selected rows or columns from a file:

```
% Create a file with values from 1 to 9
fid = fopen('nine.bin', 'w');
alldata = reshape([1:9],3,3);
fwrite(fid, alldata);
fclose(fid);
```

```
% Read the first six values into two columns
fid = fopen('nine.bin');
two_cols = fread(fid, [3, 2]);

% Return to the beginning of the file
frewind(fid);

% Read two values at a time, skip one
% Returns six values into two rows
% (first two rows of 'alldata')

two_rows = fread(fid, [2, 3], '2*uint8', 1);

% Close the file
fclose(fid);
```

Specify *machineformat* to read separate digits of binary coded decimal (BCD) values correctly:

```
% Create a file with BCD values
str = ['AB'; 'CD'; 'EF'; 'FA'];

fid = fopen('bcd.bin', 'w');
fwrite(fid, hex2dec(str), 'ubit8');
fclose(fid);

% If you read one byte at a time,
% no need to specify machine format
fid = fopen('bcd.bin');

onebyte = fread(fid, 4, '*ubit8');
disp('Correct data, read with ubit8:')
disp(dec2hex(onebyte))

% However, if you read 4 bits on a little-endian
% system, your results appear in the wrong order
```

```
frewind(fid);     % return to beginning of file

part_err = fread(fid, 8, '*ubit4');
disp('Incorrect data on little-endian systems, ubit4:')
disp(dec2hex(part_err))

% Specify a big-endian format for correct results
frewind(fid);

part_corr = fread(fid, 8, '*ubit4', 'ieee-be');
disp('Correct result, ubit4:')
disp(dec2hex(part_corr))

fclose(fid);
```

See Also

fclose | ferror | fgetl | fgets | fopen | fscanf | fprintf | fwrite | fseek | ftell | feof

How To

- "Reading Portions of a File"
- "Reading Files Created on Other Systems"

fread (serial)

Purpose Read binary data from device

Syntax

```
A = fread(obj)
A = fread(obj,size,'precision')
[A,count] = fread(...)
[A,count,msg] = fread(...)
```

Description

`A = fread(obj)` and A = fread(obj,size) read binary data from the device connected to the serial port object, `obj`, and returns the data to `A`. The maximum number of values to read is specified by `size`. If `size` is not specified, the maximum number of values to read is determined by the object's `InputBufferSize` property. Valid options for `size` are:

`n`	Read at most `n` values into a column vector.
`[m,n]`	Read at most m-by-n values filling an m–by–n matrix in column order.

`size` cannot be `inf`, and an error is returned if the specified number of values cannot be stored in the input buffer. You specify the size, in bytes, of the input buffer with the `InputBufferSize` property. A value is defined as a byte multiplied by the *`precision`* (see below).

`A = fread(obj,`*`size`*`,'`*`precision`*`')` reads binary data with precision specified by *`precision`*.

`precision` controls the number of bits read for each value and the interpretation of those bits as integer, floating-point, or character values. If *`precision`* is not specified, `uchar` (an 8-bit unsigned character) is used. By default, numeric values are returned in double-precision arrays. The supported values for *`precision`* are listed below in Tips.

`[A,count] = fread(...)` returns the number of values read to `count`.

`[A,count,msg] = fread(...)` returns a warning message to `msg` if the read operation was unsuccessful.

Tips

Before you can read data from the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read, each time `fread` is issued.

If you use the `help` command to display help for `fread`, then you need to supply the pathname shown below.

```
help serial/fread
```

Rules for Completing a Binary Read Operation

A read operation with `fread` blocks access to the MATLAB command line until:

- The specified number of values are read.
- The time specified by the `Timeout` property passes.

Note The `Terminator` property is not used for binary read operations.

Supported Precisions

The supported values for *`precision`* are listed below.

Data Type	Precision	Interpretation
Character	`uchar`	8-bit unsigned character
	`schar`	8-bit signed character
	`char`	8-bit signed or unsigned character

Data Type	Precision	Interpretation
Integer	int8	8-bit integer
	int16	16-bit integer
	int32	32-bit integer
	uint8	8-bit unsigned integer
	uint16	16-bit unsigned integer
	uint32	32-bit unsigned integer
	short	16-bit integer
	int	32-bit integer
	long	32- or 64-bit integer
	ushort	16-bit unsigned integer
	uint	32-bit unsigned integer
	ulong	32- or 64-bit unsigned integer
Floating-point	single	32-bit floating point
	float32	32-bit floating point
	float	32-bit floating point
	double	64-bit floating point
	float64	64-bit floating point

See Also

`fgetl` | `fgets` | `fopen` | `fscanf` | `BytesAvailable` | `BytesAvailableFcn` | `InputBufferSize` | `Status` | `Terminator` | `ValuesReceived`

Purpose

(Will be removed) Facets referenced by only one simplex

Note `freeBoundary(TriRep)` will be removed in a future release. Use `freeBoundary(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
FF = freeBoundary(TR)
[FF XF] = freeBoundary(TR)
```

Description

`FF = freeBoundary(TR)` returns a matrix `FF` that represents the free boundary facets of the triangulation. A facet is on the free boundary if it is referenced by only one simplex (triangle/tetrahedron, etc). `FF` is of size `m`-by-`n`, where `m` is the number of boundary facets and `n` is the number of vertices per facet. The vertices of the facets index into the array of points representing the vertex coordinates `TR.X`. The array `FF` could be empty as in the case of a triangular mesh representing the surface of a sphere.

`[FF XF] = freeBoundary(TR)` returns a matrix of free boundary facets

Input Arguments

`TR`	Triangulation representation.

Output Arguments

`FF`	`FF` that has vertices defined in terms of a compact array of coordinates `XF`.
`XF`	`XF` is of size `m`-by-`ndim` where `m` is the number of free facets, and `ndim` is the dimension of the space where the triangulation resides

Definitions

A *simplex* is a triangle/tetrahedron or higher-dimensional equivalent. A *facet* is an edge of a triangle or a face of a tetrahedron.

Examples

Example 1

Use `TriRep` to compute the boundary triangulation of an imported triangulation.

Load a 3-D triangulation:

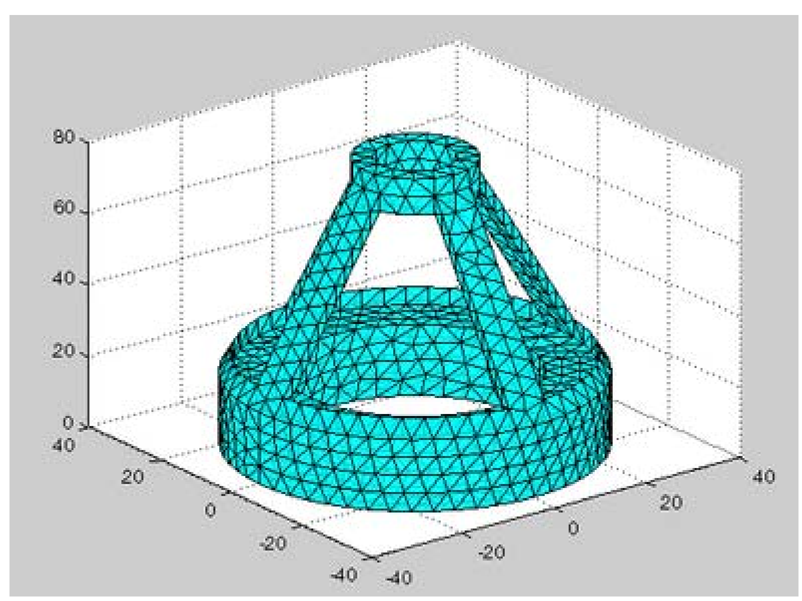
```
load tetmesh;
trep = TriRep(tet, X);
```

Compute the boundary triangulation:

```
 [tri xf] = freeBoundary(trep);
```

Plot the boundary triangulation:

```
trisurf(tri, xf(:,1),xf(:,2),xf(:,3), ...
     'FaceColor','cyan', 'FaceAlpha', 0.8);
```



Example 2

Perform a direct query of a 2-D triangulation created with `DelaunayTri`.

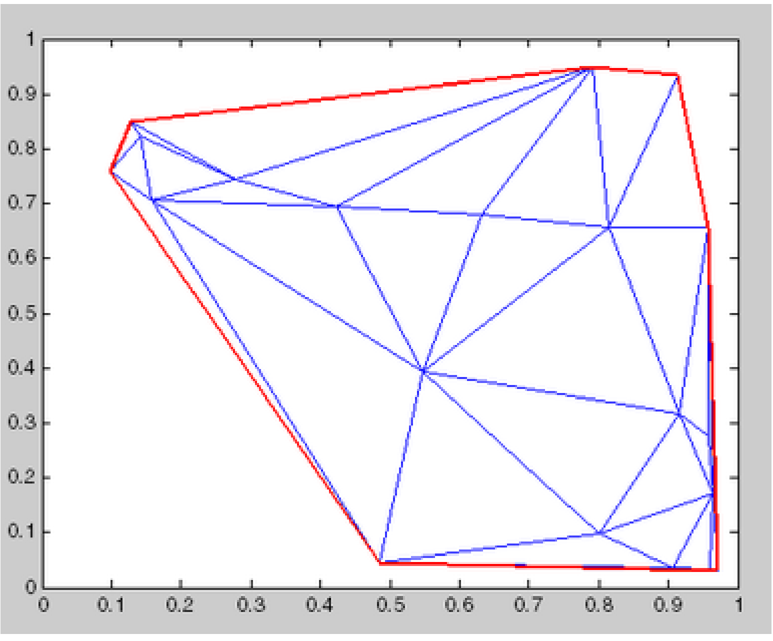
Plot the mesh:

```
x = rand(20,1);
y = rand(20,1);
dt = DelaunayTri(x,y);
fe = freeBoundary(dt)';
triplot(dt);
hold on;
```

Display the free boundary edges in red:

```
plot(x(fe), y(fe), '-r', 'LineWidth',2) ;
hold off;
```

In this instance the free edges correspond to the convex hull of `(x, y)`.



See Also

delaunayTriangulation | convexHull | featureEdges | faceNormal | triangulation

Purpose

Frequency spacing for frequency response

Syntax

```
[f1,f2] = freqspace(n)
[f1,f2] = freqspace([m n])
[x1,y1] = freqspace(...,'meshgrid')
f = freqspace(N)
f = freqspace(N,'whole')
```

Description

`freqspace` returns the implied frequency range for equally spaced frequency responses. `freqspace` is useful when creating desired frequency responses for various one- and two-dimensional applications.

`[f1,f2] = freqspace(n)` returns the two-dimensional frequency vectors `f1` and `f2` for an `n`-by-`n` matrix.

For `n` odd, both `f1` and `f2` are `[-n+1:2:n-1]/n`.

For `n` even, both `f1` and `f2` are `[-n:2:n-2]/n`.

`[f1,f2] = freqspace([m n])` returns the two-dimensional frequency vectors `f1` and `f2` for an `m`-by-`n` matrix.

`[x1,y1] = freqspace(...,'meshgrid')` is equivalent to

```
[f1,f2] = freqspace(...);
[x1,y1] = meshgrid(f1,f2);
```

`f = freqspace(N)` returns the one-dimensional frequency vector `f` assuming `N` evenly spaced points around the unit circle. For `N` even or odd, `f` is `(0:2/N:1)`. For `N` even, `freqspace` therefore returns `(N+2)/2` points. For `N` odd, it returns `(N+1)/2` points.

`f = freqspace(N,'whole')` returns `N` evenly spaced points around the whole unit circle. In this case, `f` is `0:2/N:2*(N-1)/N`.

See Also

`meshgrid`

frewind

Purpose

Move file position indicator to beginning of open file

Syntax

`frewind(fileID)`

Description

`frewind(fileID)` sets the file position indicator to the beginning of a file. *`fileID`* is an integer file identifier obtained from `fopen`.

If the file is on a tape device and the rewind operation fails, `frewind` does not return an error message.

Alternatives

`frewind(fileID)` is equivalent to:

```
fseek(fileID, 0, 'bof');
```

See Also

`fclose` | `feof` | `ferror` | `fopen` | `fseek` | `ftell` | `fscanf` | `fprintf` | `fread` | `fwrite`

Purpose

Read data from text file

Syntax

```
A = fscanf(fileID, format)
A = fscanf(fileID, format, sizeA)
[A, count] = fscanf(...)
```

Description

`A = fscanf(fileID, format)` reads and converts data from a text file into array *A* in column order. To convert, `fscanf` uses the *format* and the encoding scheme associated with the file. To set the encoding scheme, use `fopen`. The `fscanf` function reapplies the *format* throughout the entire file, and positions the file pointer at the end-of-file marker. If `fscanf` cannot match the *format* to the data, it reads only the portion that matches into *A* and stops processing.

`A = fscanf(fileID, format, sizeA)` reads *sizeA* elements into *A*, and positions the file pointer after the last element read. *sizeA* can be an integer, or can have the form `[m,n]`.

`[A, count] = fscanf(...)` returns the number of elements that `fscanf` successfully reads.

Tips

- Format specifiers for the reading functions `sscanf` and `fscanf` differ from the formats for the writing functions `sprintf` and `fprintf`. The reading functions do not support a precision field. The width field specifies a minimum for writing but a maximum for reading.

Input Arguments

fileID

Integer file identifier obtained from `fopen`.

format

String enclosed in single quotation marks that describes each type of element (field). Includes one or more of the following specifiers.

Field Type	Specifier	Details
Integer, signed	`%d`	Base 10
	`%i`	Base determined from the values. Defaults to base 10. If initial digits are `0x` or `0X`, it is base 16. If initial digit is `0`, it is base 8.
	`%ld` or `%li`	64-bit values, base 10, 8, or 16
Integer, unsigned	`%u`	Base 10
	`%o`	Base 8 (octal)
	`%x`	Base 16 (hexadecimal)
	`%lu`, `%lo`, `%lx`	64-bit values, base 10, 8, or 16
Floating-point number	`%f`	Floating-point fields can contain any of the following (not case sensitive): `Inf`, `-Inf`, `NaN`, or `-NaN`.
	`%e`	
	`%g`	
Character string	`%s`	Read series of characters, until find white space.
	`%c`	Read any single character, including white space. (To read multiple characters, specify field length.)
	`%[...]`	Read only characters in the brackets, until the first nonmatching character or white space.

Optionally:

- To skip fields, insert an asterisk (`*`) after the percent sign (`%`). For example, to skip integers, specify `%*d`.

- To specify the maximum width of a field, insert a number. For example, `%10c` reads exactly 10 characters at a time, including white space.
- To skip a specific set of characters, insert the literal characters in the *format*. For example, to read only the floating-point number from `'pi=3.14159'`, specify a *format* of `'pi=%f'`.

sizeA

Dimensions of the output array *A*. Specify in one of the following forms:

`inf`	Read to the end of the file. (default)
n	Read at most *n* elements.
[*m*,*n*]	Read at most *m***n* elements in column order. *n* can be `inf`, but *m* cannot.

When the *format* includes `%s`, *A* can contain more than *n* columns. *n* refers to elements, not characters.

Output Arguments

A

An array. If the *format* includes:

- Only numeric specifiers, *A* is numeric. If *format* includes only 64-bit signed integer specifiers, *A* is of class `int64`. Similarly, if *format* includes only 64-bit unsigned integer specifiers, *A* is of class `uint64`. Otherwise, *A* is of class `double`. If *sizeA* is `inf` or *n*, then *A* is a column vector. If the input contains fewer than *sizeA* elements, MATLAB pads *A* with zeros.
- Only character or string specifiers (`%c` or `%s`), *A* is a character array. If *sizeA* is `inf` or *n*, *A* is a row vector. If the input contains fewer than *sizeA* characters, MATLAB pads *A* with `char(0)`.
- A combination of numeric and character specifiers, *A* is numeric, of class `double`. MATLAB converts each character to its numeric equivalent. This conversion occurs even when the *format* explicitly skips all numeric values (for example, a *format* of `'%*d %s'`).

If MATLAB cannot match the input to the *format*, and the *format* contains both numeric and character specifiers, *A* can be numeric or character. The class of *A* depends on the values MATLAB reads before processing stops.

count

The number of elements `fscanf` reads into *A*.

Examples

Read the contents of a file. `fscanf` reuses the *format* throughout the file, so you do not need a control loop:

```
% Create a file with an exponential table
x = 0:.1:1;
y = [x; exp(x)];

fid = fopen('exp.txt', 'w');
fprintf(fid, '%6.2f %12.8f\n', y);
fclose(fid);

% Read the data, filling A in column order
% First line of the file:
%    0.00    1.00000000

fid = fopen('exp.txt');
A = fscanf(fid, '%g %g', [2 inf]);
fclose(fid);

% Transpose so that A matches
% the orientation of the file
A = A';
```

Skip specific characters in a file, and return only numeric values:

```
% Create a file with temperatures
tempstr = '78 F 72 F 64 F 66 F 49 F';
```

```
fid = fopen('temperature.dat', 'w+');
fprintf(fid, '%s', tempstr);

% Return to the beginning of the file
frewind(fid);

% Read the numbers in the file, skipping the units
% num_temps is a numeric column vector

degrees = char(176);
num_temps = fscanf(fid, ['%d' degrees 'F']);

fclose(fid);
```

See Also

fclose | ferror | fgetl | fgets | fopen | fprintf | fread | fwrite | sscanf | textscan

How To

- “Reading Data in a Formatted Pattern”
- “Opening Files with Different Character Encodings”

fscanf (serial)

Purpose

Read data from device, and format as text

Syntax

```
A = fscanf(obj)
A = fscanf(obj,'format')
A = fscanf(obj,'format',size)
[A,count] = fscanf(...)
[A,count,msg] = fscanf(...)
```

Description

`A = fscanf(obj)` reads data from the device connected to the serial port object, `obj`, and returns it to `A`. The data is converted to text using the `%c` format.

`A = fscanf(obj,'format')` reads data and converts it according to *`format`*. *`format`* is a C language conversion specification. Conversion specifications involve the `%` character and the conversion characters d, i, o, u, x, X, f, e, E, g, G, c, and s. Refer to the `sscanf` file I/O format specifications or a C manual for more information.

`A = fscanf(obj,'format',size)` reads the number of values specified by `size`. Valid options for `size` are:

`n`	Read at most `n` values into a column vector.
`[m,n]`	Read at most m-by-n values filling an m–by–n matrix in column order.

`size` cannot be `inf`, and an error is returned if the specified number of values cannot be stored in the input buffer. If `size` is not of the form `[m,n]`, and a character conversion is specified, then `A` is returned as a row vector. You specify the size, in bytes, of the input buffer with the `InputBufferSize` property. An ASCII value is one byte.

`[A,count] = fscanf(...)` returns the number of values read to `count`.

`[A,count,msg] = fscanf(...)` returns a warning message to `msg` if the read operation did not complete successfully.

Tips

Before you can read data from the device, it must be connected to obj with the fopen function. A connected serial port object has a Status property value of open. An error is returned if you attempt to perform a read operation while obj is not connected to the device.

If msg is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The ValuesReceived property value is increased by the number of values read – including the terminator – each time fscanf is issued.

If you use the help command to display help for fscanf, then you need to supply the pathname shown below.

```
help serial/fscanf
```

Rules for Completing a Read Operation with fscanf

A read operation with fscanf blocks access to the MATLAB command line until:

- The terminator specified by the Terminator property is read.
- The time specified by the Timeout property passes.
- The number of values specified by size is read.
- The input buffer is filled (unless size is specified)

Examples

Create the serial port object s and connect s to a Tektronix TDS 210 oscilloscope, which is displaying sine wave. This example works on a Windows platform.

```
s = serial('COM1');
fopen(s)
```

Use the fprintf function to configure the scope to measure the peak-to-peak voltage of the sine wave, return the measurement type, and return the peak-to-peak voltage.

```
fprintf(s,'MEASUREMENT:IMMED:TYPE PK2PK')
```

```
fprintf(s,'MEASUREMENT:IMMED:TYPE?')
fprintf(s,'MEASUREMENT:IMMED:VALUE?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data associated with the two query commands is automatically returned to the input buffer.

```
s.BytesAvailable
ans =
    21
```

Use `fscanf` to read the measurement type. The operation will complete when the first terminator is read.

```
meas = fscanf(s)
meas =
PK2PK
```

Use `fscanf` to read the peak-to-peak voltage as a floating-point number, and exclude the terminator.

```
pk2pk = fscanf(s,'%e',14)
pk2pk =
    2.0200
```

Disconnect `s` from the scope, and remove `s` from memory and the workspace.

```
fclose(s)
delete(s)
clear s
```

See Also

`fgetl` | `fgets` | `fopen` | `fread` | `textscan` | `BytesAvailable` | `BytesAvailableFcn` | `InputBufferSize` | `Status` | `Terminator` | `Timeout`

Purpose

Move to specified position in file

Syntax

```
fseek(fileID, offset, origin)
status = fseek(fileID, offset, origin)
```

Description

`fseek(fileID, offset, origin)` sets the file position indicator *`offset`* bytes from *`origin`* in the specified file.

`status = fseek(fileID, offset, origin)` returns `0` when the operation is successful. Otherwise, it returns `-1`.

Input Arguments

fileID

Integer file identifier obtained from `fopen`.

offset

Number of bytes to move from `origin`. Can be positive, negative, or zero. The *n* bytes of a given file are in positions `0` through *n*`-1`.

origin

Starting location in the file:

`'bof'` or `-1`	Beginning of file
`'cof'` or `0`	Current position in file
`'eof'` or `1`	End of file

Examples

Copy 5 bytes from the file `test1.dat`, starting at the tenth byte, and append to the end of `test2.dat`:

```
% Create files test1.dat and test2.dat
% Each character uses 8 bits (1 byte)

fid1 = fopen('test1.dat', 'w+');
fwrite(fid1, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ');
```

```
fid2 = fopen('test2.dat', 'w+');
fwrite(fid2, 'Second File');

% Seek to the 10th byte ('J'), read 5
fseek(fid1, 9, 'bof');
A = fread(fid1, 5, 'uint8=>char');
fclose(fid1);

% Append to test2.dat
fseek(fid2, 0, 'eof');
fwrite(fid2, A);
fclose(fid2);
```

Alternatives

To move to the beginning of a file, call

```
frewind(fileID)
```

This call is identical to

```
fseek(fileID, 0, 'bof')
```

See Also

fclose | feof | ferror | fopen | frewind | ftell | fscanf | fprintf | fread | fwrite

How To

- “Reading Portions of a File”

Purpose

Position in open file

Syntax

`position = ftell(fileID)`

Description

position = ftell(*fileID*) returns the current position in the specified file. *position* is a zero-based integer that indicates the number of bytes from the beginning of the file. If the query is unsuccessful, *position* is -1. *fileID* is an integer file identifier obtained from fopen.

See Also

fclose | feof | ferror | fopen | frewind | fseek | fscanf | fprintf | fread | fwrite

Purpose

Connect to FTP server

Description

Connect to an FTP server by calling the `ftp` function, which creates an FTP object. Perform file operations using methods on the FTP object, such as `mput` and `mget`. When you finish accessing the server, call the `close` method to close the connection.

Construction

`f = ftp(host,username,password)` connects to the FTP server `host` and creates FTP object `f`. If the host supports anonymous connections, you can use the `host` argument alone. To specify an alternate port, separate it from `host` with a colon (`:`).

Input Arguments

host

String enclosed in single quotation marks that specifies the FTP server.

username

String enclosed in single quotation marks that specifies your user name for the FTP server.

password

String enclosed in single quotation marks that specifies your password for the FTP server. Because FTP is not a secure protocol, others can see your user name and password.

Methods

ascii	Set FTP transfer type to ASCII
binary	Set FTP transfer type to binary
cd	Change or view current folder on FTP server
close	Close connection to FTP server
delete	Remove file on FTP server

dir	View contents of folder on FTP server
mget	Download files from FTP server
mkdir	Create folder on FTP server
mput	Upload file or folder to FTP server
rename	Rename file on FTP server
rmdir	Remove folder on FTP server

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Connect to the MathWorks FTP server, and display the FTP object:

```
mw=ftp('ftp.mathworks.com');
disp(mw)
```

MATLAB returns:

```
FTP Object
  host: ftp.mathworks.com
  user: anonymous
   dir: /
  mode: binary
```

Connect to port 34 (not supported at `ftp.mathworks.com`, so this code returns an error):

```
mw=ftp('ftp.mathworks.com:34')
```

Modify the following code to connect to a host that requires a password:

```
test=ftp('ftp.testsite.com','myname','mypassword')
```

Algorithms

The `ftp` function is based on code from the Apache Jakarta Project.

See Also

`urlwrite`

How To

- "Specify Proxy Server Settings for Connecting to the Internet"

Purpose

Convert sparse matrix to full matrix

Syntax

```
A = full(S)
```

Description

`A = full(S)` converts a sparse matrix `S` to full storage organization. If `S` is a full matrix, it is left unchanged. If `A` is full, `issparse(A)` is `0`.

Tips

If `X` is an `m`-by-`n` matrix with `nz` nonzero elements then `full(X)` requires space to store `m*n` elements. On the other hand, `sparse(X)` requires space to store `nz` elements and `(nz+n+1)` integers.

The density of a matrix (`nnz(X)/numel(X)`) determines whether or not it is more efficient to store the matrix as sparse or full. The exact crossover point depends on the matrix class as well as the platform. For example, in 32-bit MATLAB, a double sparse matrix with less than about 2/3 density will require less space than the same matrix in full storage. In 64-bit MATLAB, however, double matrices with less than half of their elements nonzero are more efficient to store as sparse matrices.

Examples

Here is an example of a sparse matrix with a density of about two-thirds. `sparse(S)` and `full(S)` require about the same number of bytes of storage.

```
S = sparse(+(rand(200,200) < 2/3));
A = full(S);
whos
Name    Size    Bytes     Class
  A    200X200 320000 double array
  S    200X200 318432 double array (sparse)
```

See Also

`issparse` | `sparse`

fullfile

Purpose

Build full file name from parts

Syntax

```
f = fullfile(filepart1,...,filepartN)
```

Description

`f = fullfile(filepart1,...,filepartN)` builds a full file specification, `f`, from the folders and file names specified. `fullfile` inserts platform-dependent file separators where necessary, and does not add a trailing file separator. The output of `fullfile` is conceptually equivalent to

```
f = [filepart1 filesep filepart2 filesep ... filesep filepartN]
```

Input Arguments

filepart1,...,filepartN - Folder and file names

strings | cell arrays of strings

Folder and file names, specified as strings and cell arrays of strings. Any nonscalar cell arrays must be of the same size.

Example: `'folder1','folder2','myfile.m'`

Example:
`{'folder1';'folder2'},{'subfolder1';'subfolder2'},'myfile.m'`

Examples

Create a Full File Name

```
f = fullfile('myfolder','mysubfolder','myfile.m')
```

```
f =
myfolder\mysubfolder\myfile.m
```

`fullfile` returns a string containing the full path to the file. On Windows platforms, the file separator character is a backslash, `\`. On other platforms, the file separator might be a different character.

Create Paths to Multiple Files

```
f = fullfile(toolboxdir('matlab'),'iofun',{'filesep.m';'fullfile.m'});
```

`fullfile` returns a cell array containing a path to the file `filesep.m`, and a path to the file `fullfile.m`.

Create a Path to a Folder

```
f = fullfile(matlabroot,'toolbox','matlab',filesep);
```

`fullfile` does not trim a leading or trailing `filesep`.

See Also

`fileparts` | `filesep` | `path` | `pathsep` | `genpath`

Purpose

Construct function name string from function handle

Syntax

```
func2str(fhandle)
```

Description

`func2str(fhandle)` constructs a string `s` that holds the name of the function to which the function handle `fhandle` belongs.

When you need to perform a string operation, such as compare or display, on a function handle, you can use `func2str` to construct a string bearing the function name.

The `func2str` command does not operate on nonscalar function handles. Passing a nonscalar function handle to `func2str` results in an error.

Tips

Any variables and their values originally stored in a function handle when it was created are lost if you convert the function handle to a string and back again using the `func2str` and `str2func` functions.

Examples

Example 1

Convert a `sin` function handle to a string:

```
fhandle = @sin;

func2str(fhandle)
ans =
    sin
```

Example 2

The `catcherr` function shown here accepts function handle and data arguments and attempts to evaluate the function through its handle. If the function fails to execute, `catcherr` uses `sprintf` to display an error message giving the name of the failing function. The function name must be a string for `sprintf` to display it. The code derives the function name from the function handle using `func2str`:

```
function catcherr(func, data)
try
```

```
    ans = func(data);
    disp('Answer is:');
    ans
catch
    disp(sprintf('Error executing function ''%s''\n', ...
        func2str(func)))
end
```

The first call to `catcherr` passes a handle to the `round` function and a valid data argument. This call succeeds and returns the expected answer. The second call passes the same function handle and an improper data type (a MATLAB structure). This time, `round` fails, causing `catcherr` to display an error message that includes the failing function name:

```
catcherr(@round, 5.432)
ans =
Answer is 5

xstruct.value = 5.432;
catcherr(@round, xstruct)
Error executing function "round"
```

See Also

`function_handle` | `str2func` | `functions`

function

Purpose

Declare function name, inputs, and outputs

Syntax

```
function [y1,...,yN] = myfun(x1,...,xM)
```

Description

`function [y1,...,yN] = myfun(x1,...,xM)` declares a function named `myfun` that accepts inputs `x1,...,xM` and returns outputs `y1,...,yN`. This declaration statement must be the first executable line of the function.

Save the function code in a text file with a `.m` extension. The name of the file should match the name of the first function in the file. Valid function names begin with an alphabetic character, and can contain letters, numbers, or underscores.

You can declare multiple local functions within the same file, or nest functions. If any function in a file contains a nested function, all functions in the file must use the `end` keyword to indicate the end of the function. Otherwise, the `end` keyword is optional.

Examples

Function with One Output

Define a function in a file named `average.m` that accepts an input vector, calculates the average of the values, and returns a single result.

```
function y = average(x)
if ~isvector(x)
    error('Input must be a vector')
end
y = sum(x)/length(x);
```

Call the function from the command line.

```
z = 1:99;
average(z)
```

```
ans =
    50
```

Function with Multiple Outputs

Define a function in a file named `stat.m` that returns the mean and standard deviation of an input vector.

```
function [m,s] = stat(x)
n = length(x);
m = sum(x)/n;
s = sqrt(sum((x-m).^2/n));
```

Call the function from the command line.

```
values = [12.7, 45.4, 98.9, 26.6, 53.1];
[ave,stdev] = stat(values)
```

```
ave =
   47.3400
stdev =
   29.4124
```

Multiple Functions in a File

Define two functions in a file named `stat2.m`, where the first function calls the second.

```
function [m,s] = stat2(x)
n = length(x);
m = avg(x,n);
s = sqrt(sum((x-m).^2/n));
end

function m = avg(x,n)
m = sum(x)/n;
end
```

Function `avg` is a *local function*. Local functions are only available to other functions within the same file.

Call function `stat2` from the command line.

```
values = [12.7, 45.4, 98.9, 26.6, 53.1];
[ave,stdev] = stat2(values)

ave =
   47.3400
stdev =
   29.4124
```

See Also

nargin | nargout | pcode | return | varargin | varargout | what | which

Related Examples

- "Create Functions in Files"

Concepts

- "Local Functions"
- "Nested Functions"
- "Base and Function Workspaces"
- "Function Precedence Order"

Purpose

Handle used in calling functions indirectly

Syntax

```
handle = @functionname
handle = @(arglist)anonymous_function
```

Description

`handle = @functionname` returns a handle to the specified MATLAB function.

A function handle is a MATLAB value that provides a means of calling a function indirectly. You can pass function handles in calls to other functions (often called *function functions*). You can also store function handles in data structures for later use (for example, as Handle Graphics callbacks). A function handle is one of the standard MATLAB data types.

At the time you create a function handle, the function you specify must be on the MATLAB path and in the current scope of the code creating the handle. For example, you can create a handle to a local function as long as you do so from within the file that defines that local function. This condition does not apply when you evaluate the function handle. You can, for example, execute a local function from a separate (out-of-scope) file using a function handle. This requires that the handle was created by the local function (in-scope).

`handle = @(arglist)anonymous_function` constructs an anonymous function and returns a `handle` to that function. The body of the function, to the right of the parentheses, is a single MATLAB statement or command. `arglist` is a comma-separated list of input arguments. Execute the function by calling it by means of the function handle, `handle`.

Tips

The function handle is a standard MATLAB data type. As such, you can manipulate and operate on function handles in the same manner as on other MATLAB data types. This includes using function handles in structures and cell arrays:

```
S.a = @sin;  S.b = @cos;  S.c = @tan;
C = {@sin, @cos, @tan};
```

However, standard matrices or arrays of function handles are not supported:

```
A = [@sin, @cos, @tan];        % This is not supported
```

For nonoverloaded functions, local functions, and private functions, a function handle references just the one function specified in the `@functionname` syntax. When you evaluate an overloaded function by means of its handle, the arguments the handle is evaluated with determine the actual function that MATLAB dispatches to.

Use `isa(h, 'function_handle')` to see if variable `h` is a function handle.

Examples

Example 1 — Constructing a Handle to a Named Function

The following example creates a function handle for the `humps` function and assigns it to the variable `fhandle`.

```
fhandle = @humps;
```

Pass the handle to another function in the same way you would pass any argument. This example passes the function handle just created to `fminbnd`, which then minimizes over the interval `[0.3, 1]`.

```
x = fminbnd(fhandle, 0.3, 1)
x =
    0.6370
```

The `fminbnd` function evaluates the `@humps` function handle. A small portion of the `fminbnd` file is shown below. In line 1, the `funfcn` input parameter receives the function handle `@humps` that was passed in. The statement, in line 113, evaluates the handle.

```
1    function [xf,fval,exitflag,output] = ...
         fminbnd(funfcn,ax,bx,options,varargin)
             .
             .
             .
```

```
113  fx = funfcn(x,varargin{:});
```

Example 2 — Constructing a Handle to an Anonymous Function

The statement below creates an anonymous function that finds the square of a number. When you call this function, MATLAB assigns the value you pass in to variable `x`, and then uses `x` in the equation `x.^2`:

```
sqr = @(x) x.^2;
```

The `@` operator constructs a function handle for this function, and assigns the handle to the output variable `sqr`. As with any function handle, you execute the function associated with it by specifying the variable that contains the handle, followed by a comma-separated argument list in parentheses. The syntax is

```
fhandle(arg1, arg2, ..., argN)
```

To execute the `sqr` function defined above, type

```
a = sqr(5)
a =
   25
```

Because `sqr` is a function handle, you can pass it in an argument list to other functions. The code shown here passes the `sqr` anonymous function to the MATLAB `integral` function to compute its integral from zero to one:

```
integral(sqr, 0, 1)
ans =
   0.3333
```

Example 3 — Using an Array of Function Handles

This example creates a structure array of function handles `S` and then applies each handle in the array to the output of a `linspace` calculation in one operation using `structfun`:

```
S.a = @sin;  S.b = @cos;  S.c = @tan;
```

```
structfun(@(x)x(linspace(1,4,3)), S, 'UniformOutput', false)
ans =
    a: [0.8415 0.5985 -0.7568]
    b: [0.5403 -0.8011 -0.6536]
    c: [1.5574 -0.7470 1.1578]
```

See Also

str2func | func2str | functions | isa

Purpose

Information about function handle

Syntax

```
S = functions(funhandle)
```

Description

`S = functions(funhandle)` returns, in MATLAB structure `S`, the function name, type, filename, and other information for the function handle stored in the variable `funhandle`.

`functions` does not operate on nonscalar function handles. Passing a nonscalar function handle to `functions` results in an error.

Caution The `functions` function is provided for querying and debugging purposes. Because its behavior may change in subsequent releases, you should not rely upon it for programming purposes.

This table lists the standard fields of the return structure.

Field Name	Field Description
`function`	Function name
`type`	Function type (e.g., simple, overloaded)
`file`	The file to be executed when the function handle is evaluated with a nonoverloaded data type

Examples

Example 1

To obtain information on a function handle for the `poly` function, type

```
f = functions(@poly)
f =
    function: 'poly'
        type: 'simple'
        file:  '$matlabroot\toolbox\matlab\polyfun\poly.m'
```

(The term `$matlabroot` used in this example stands for the file specification of the directory in which MATLAB software is installed for your system. Your output will display this file specification.)

Access individual fields of the returned structure using dot selection notation:

```
f.type
ans =
   simple
```

Example 2

The function `get_handles` returns function handles for a local function and private function in output arguments `l` and `p` respectively:

```
function [l, p] = get_handles
l = @mylocfun;
p = @myprivatefun;
%
function mylocfun
disp 'Executing local function mylocfun'
```

Call `get_handles` to obtain the two function handles, and then pass each to the `functions` function. MATLAB returns information in a structure having the fields `function`, `type`, `file`, and `parentage`. The `file` field contains the file specification for the local or private function:

```
[floc fprv] = get_handles;

functions(floc)
ans =
     function: 'mylocfun'
         type: 'scopedfunction'
         file: 'c:\matlab\get_handles.m'
    parentage: {'mylocfun'  'get_handles'}

functions(fprv)
ans =
```

```
     function: 'myprivatefun'
         type: 'scopedfunction'
         file: 'c:\matlab\private\myprivatefun.m'
    parentage: {'myprivatefun'}
```

Example 3

In this example, the function `get_handles_nested.m` contains a nested function `nestfun`. This function has a single output which is a function handle to the nested function:

```
function handle = get_handles_nested(A)
nestfun(A);

   function y = nestfun(x)
   y = x + 1;
   end

handle = @nestfun;
end
```

Call this function to get the handle to the nested function. Use this handle as the input to `functions` to return the information shown here. Note that the `function` field of the return structure contains the names of the nested function and the function in which it is nested in the format. Also note that `functions` returns a `workspace` field containing the variables that are in context at the time you call this function by its handle:

```
fh = get_handles_nested(5);

fhinfo = functions(fh)
fhinfo =
     function: 'get_handles_nested/nestfun'
         type: 'nested'
         file: 'c:\matlab\get_handles_nested.m'
    workspace: [1x1 struct]

fhinfo.workspace
```

```
ans =
    handle: @get_handles_nested/nestfun
         A: 5
```

See Also function_handle

Purpose

Evaluate general matrix function

Syntax

```
F = funm(A,fun)
F = funm(A,fun,options)
F = funm(A,fun,options,p1,p2,...)
[F,exitflag] = funm(...)
[F,exitflag,output] = funm(...)
```

Description

`F = funm(A,fun)` evaluates the user-defined function `fun` at the square matrix argument `A`. `F = fun(x,k)` must accept a vector `x` and an integer `k`, and return a vector `f` of the same size of `x`, where `f(i)` is the `k`th derivative of the function `fun` evaluated at `x(i)`. The function represented by fun must have a Taylor series with an infinite radius of convergence, except for `fun = @log`, which is treated as a special case.

You can also use `funm` to evaluate the special functions listed in the following table at the matrix `A`.

Function	Syntax for Evaluating Function at Matrix A
`exp`	`funm(A, @exp)`
`log`	`funm(A, @log)`
`sin`	`funm(A, @sin)`
`cos`	`funm(A, @cos)`
`sinh`	`funm(A, @sinh)`
`cosh`	`funm(A, @cosh)`

For matrix square roots, use `sqrtm(A)` instead. For matrix exponentials, which of `expm(A)` or `funm(A, @exp)` is the more accurate depends on the matrix `A`.

The function represented by `fun` must have a Taylor series with an infinite radius of convergence. The exception is `@log`, which is treated as a special case. “Parameterizing Functions” explains how to provide additional parameters to the function `fun`, if necessary.

`F = funm(A,fun,options)` sets the algorithm's parameters to the values in the structure `options`.

The following table lists the fields of `options`.

Field	Description	Values
`options.Display`	Level of display	`'off'` (default), `'on'`, `'verbose'`
`options.TolBlk`	Tolerance for blocking Schur form	Positive scalar. The default is `0.1`.
`options.TolTay`	Termination tolerance for evaluating the Taylor series of diagonal blocks	Positive scalar. The default is `eps`.
`options.MaxTerms`	Maximum number of Taylor series terms	Positive integer. The default is `250`.
`options.MaxSqrt`	When computing a logarithm, maximum number of square roots computed in inverse scaling and squaring method.	Positive integer. The default is `100`.
`options.Ord`	Specifies the ordering of the Schur form `T`.	A vector of length `length(A)`. `options.Ord(i)` is the index of the block into which `T(i,i)` is placed. The default is `[]`.

`F = funm(A,fun,options,p1,p2,...)` passes extra inputs `p1,p2,...` to the function.

`[F,exitflag] = funm(...)` returns a scalar `exitflag` that describes the exit condition of `funm`. `exitflag` can have the following values:

- `0` — The algorithm was successful.

- `1` — One or more Taylor series evaluations did not converge, or, in the case of a logarithm, too many square roots are needed. However, the computed value of `F` might still be accurate.

`[F,exitflag,output] = funm(...)` returns a structure `output` with the following fields:

Field	Description
`output.terms`	Vector for which `output.terms(i)` is the number of Taylor series terms used when evaluating the *i*th block, or, in the case of the logarithm, the number of square roots of matrices of dimension greater than 2.
`output.ind`	Cell array for which the `(i,j)` block of the reordered Schur factor `T` is `T(output.ind{i}, output.ind{j})`.
`output.ord`	Ordering of the Schur form, as passed to `ordschur`
`output.T`	Reordered Schur form

If the Schur form is diagonal then `output = struct('terms',ones(n,1),'ind',{1:n})`.

Examples

Example 1

The following command computes the matrix sine of the 3-by-3 magic matrix.

```
F=funm(magic(3), @sin)

F =

   -0.3850    1.0191    0.0162
    0.6179    0.2168   -0.1844
    0.4173   -0.5856    0.8185
```

Example 2

The statements

```
S = funm(X,@sin);
C = funm(X,@cos);
```

produce the same results to within roundoff error as

```
E = expm(i*X);
C = real(E);
S = imag(E);
```

In either case, the results satisfy `S*S+C*C = I`, where `I = eye(size(X))`.

Example 3

To compute the function `exp(x) + cos(x)` at `A` with one call to `funm`, use

```
F = funm(A,@fun_expcos)
```

where `fun_expcos` is the following function.

```
function f = fun_expcos(x, k)
% Return kth derivative of exp + cos at X.
        g = mod(ceil(k/2),2);
        if mod(k,2)
           f = exp(x) + sin(x)*(-1)^g;
        else
           f = exp(x) + cos(x)*(-1)^g;
        end
```

Algorithms

The algorithm `funm` uses is described in [1].

References

[1] Davies, P. I. and N. J. Higham, "A Schur-Parlett algorithm for computing matrix functions," *SIAM J. Matrix Anal. Appl.*, Vol. 25, Number 2, pp. 464-485, 2003.

[2] Golub, G. H. and C. F. Van Loan, *Matrix Computation*, Third Edition, Johns Hopkins University Press, 1996, p. 384.

[3] Moler, C. B. and C. F. Van Loan, "Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later" *SIAM Review 20*, Vol. 45, Number 1, pp. 1-47, 2003.

See Also

`expm` | `logm` | `sqrtm` | `function_handle`

Purpose

Write data to binary file

Syntax

```
fwrite(fileID, A)
fwrite(fileID, A, precision)
fwrite(fileID, A, precision, skip)
fwrite(fileID, A, precision, skip, machineformat)
count = fwrite(...)
```

Description

`fwrite(fileID, A)` writes the elements of array *A* to a binary file in column order.

`fwrite(fileID, A, precision)` translates the values of *A* according to the form and size described by the *precision*.

`fwrite(fileID, A, precision, skip)` skips *skip* bytes before writing each value. If *precision* is `bitn` or `ubitn`, specify *skip* in bits.

`fwrite(fileID, A, precision, skip, machineformat)` writes data with the specified *machineformat*. The *skip* parameter is optional.

`count = fwrite(...)` returns the number of elements of *A* that `fwrite` successfully writes to the file.

Input Arguments

fileID

File identifier, specified as one of the following:

- A file identifier obtained from `fopen`.
- `1` for standard output (the screen).
- `2` for standard error.

A

Numeric or character array.

precision

String in single quotation marks that controls the form and size of the output. The following table shows possible values for *precision*.

Value Type	Precision	Bits (Bytes)
Integers, unsigned	uint	32 (4)
	uint8	8 (1)
	uint16	16 (2)
	uint32	32 (4)
	uint64	64 (8)
	uchar	8 (1)
	unsigned char	8 (1)
	ushort	16 (2)
	ulong	system-dependent
	ubit*n*	$1 \leq n \leq 64$
Integers, signed	int	32 (4)
	int8	8 (1)
	int16	16 (2)
	int32	32 (4)
	int64	64 (8)
	integer*1	8 (1)
	integer*2	16 (2)
	integer*4	32 (4)
	integer*8	64 (8)
	schar	8 (1)
	signed char	8 (1)
	short	16 (2)
	long	system-dependent
	bit*n*	$1 \leq n \leq 64$

Value Type	Precision	Bits (Bytes)
Floating-point numbers	`single`	`32 (4)`
	`double`	`64 (8)`
	`float`	`32 (4)`
	`float32`	`32 (4)`
	`float64`	`64 (8)`
	`real*4`	`32 (4)`
	`real*8`	`64 (8)`
Characters	`char*1`	`8 (1)`
	`char`	Depends on the encoding scheme associated with the file. Set encoding with `fopen`.

On 32-bit systems, `long` and `ulong` are 32 bits. On 64-bit systems, the sizes of `long` and `ulong` correspond to the data model that the C compiler uses: 32 bits on Microsoft Windows systems, 64 bits on other 64-bit systems.

If you specify a precision of `bit`*n* or `ubit`*n*, and a value is out of range, `fwrite` sets all bits for that value.

Default: `uint8`

skip

Number of bytes to skip before writing each value. If you specify a *`precision`* of `bit`*n* or `ubit`*n*, specify *`skip`* in bits. Use this parameter to insert data into noncontiguous fields in fixed-length records.

Default: `0`

machineformat

String that specifies the order for writing bytes within the file. For `bitn` and `ubitn` precisions, specifies the order for writing bits within a byte.

Possible values are:

`'n'` or `'native'`	The byte ordering that your system uses (default)
`'b'` or `'ieee-be'`	Big-endian ordering
`'l'` or `'ieee-le'`	Little-endian ordering
`'s'` or `'ieee-be.l64'`	Big-endian ordering, 64-bit long data type
`'a'` or `'ieee-le.l64'`	Little-endian ordering, 64-bit long data type

By default, all currently supported platforms use little-endian ordering for new files. Existing binary files can use either big- or little-endian ordering.

Examples

Create a 100-byte binary file containing the 25 elements of the 5-by-5 magic square, stored as 4-byte integers:

```
fid = fopen('magic5.bin', 'w');
fwrite(fid, magic(5), 'integer*4');
fclose(fid);
```

See Also

`fclose` | `ferror` | `fopen` | `fprintf` | `fscanf` | `fread` | `fseek` | `ftell`

How To

- “Overwriting or Appending to an Existing File”
- “Creating a File for Use on a Different System”
- “Writing and Reading Complex Numbers”

fwrite (serial)

Purpose

Write binary data to device

Syntax

```
fwrite(obj,A)
fwrite(obj,A,'precision')
fwrite(obj,A,'mode')
fwrite(obj,A,'precision','mode')
```

Description

`fwrite(obj,A)` writes the binary data `A` to the device connected to the serial port object, `obj`.

`fwrite(obj,A,'precision')` writes binary data with precision specified by *`precision`*.

`precision` controls the number of bits written for each value and the interpretation of those bits as integer, floating-point, or character values. If *`precision`* is not specified, `uchar` (an 8-bit unsigned character) is used. The supported values for *`precision`* are listed below in Tips.

`fwrite(obj,A,'mode')` writes binary data with command line access specified by *`mode`*. If *`mode`* is `sync`, `A` is written synchronously and the command line is blocked. If *`mode`* is `async`, `A` is written asynchronously and the command line is not blocked. If *`mode`* is not specified, the write operation is synchronous.

`fwrite(obj,A,'precision','mode')` writes binary data with precision specified by *`precision`* and command line access specified by *`mode`*.

Tips

Before you can write data to the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a write operation while `obj` is not connected to the device.

The `ValuesSent` property value is increased by the number of values written each time `fwrite` is issued.

An error occurs if the output buffer cannot hold all the data to be written. You can specify the size of the output buffer with the `OutputBufferSize` property.

If you use the `help` command to display help for `fwrite`, then you need to supply the pathname shown below.

```
help serial/fwrite
```

`fwrite` will return an error message if you set the `FlowControl` property to `hardware` on a serial object, and a hardware connection is not detected. This occurs if a device is not connected, or a connected device is not asserting that is ready to receive data. Check you remote device's status and flow control settings to see if hardware flow control is causing errors in MATLAB.

Note If you want to check to see if the device is asserting that it is ready to receive data, set the `FlowControl` to `none`. Once you connect to the device check the `PinStatus` structure for `ClearToSend`. If `ClearToSend` is `off`, there is a problem on the remote device side. If `ClearToSend` is `on`, there is a hardware `FlowControl` device prepared to receive data and you can execute `fwrite`.

Synchronous Versus Asynchronous Write Operations

By default, data is written to the device synchronously and the command line is blocked until the operation completes. You can perform an asynchronous write by configuring the *`mode`* input argument to be `async`. For asynchronous writes:

- The `BytesToOutput` property value is continuously updated to reflect the number of bytes in the output buffer.
- The callback function specified for the `OutputEmptyFcn` property is executed when the output buffer is empty.

You can determine whether an asynchronous write operation is in progress with the `TransferStatus` property.

Synchronous and asynchronous write operations are discussed in more detail in Writing Data.

Rules for Completing a Write Operation with fwrite

A binary write operation using `fwrite` completes when:

- The specified data is written.
- The time specified by the `Timeout` property passes.

> **Note** The `Terminator` property is not used with binary write operations.

Supported Precisions

The supported values for *`precision`* are listed below.

Data Type	Precision	Interpretation
Character	`uchar`	8-bit unsigned character
	`schar`	8-bit signed character
	`char`	8-bit signed or unsigned character
Integer	`int8`	8-bit integer
	`int16`	16-bit integer
	`int32`	32-bit integer
	`uint8`	8-bit unsigned integer
	`uint16`	16-bit unsigned integer
	`uint32`	32-bit unsigned integer
	`short`	16-bit integer
	`int`	32-bit integer
	`long`	32- or 64-bit integer
	`ushort`	16-bit unsigned integer
	`uint`	32-bit unsigned integer

Data Type	Precision	Interpretation
	ulong	32- or 64-bit unsigned integer
Floating-point	single	32-bit floating point
	float32	32-bit floating point
	float	32-bit floating point
	double	64-bit floating point
	float64	64-bit floating point

See Also

fopen | fprintf | BytesToOutput | OutputBufferSize | OutputEmptyFcn | Status | Timeout | TransferStatus | ValuesSent

fzero

Purpose

Root of nonlinear function

Syntax

```
x = fzero(fun,x0)
x = fzero(fun,x0,options)

x = fzero(problem)

[x,fval,exitflag,output] = fzero( ___ )
```

Description

`x = fzero(fun,x0)` tries to find a point `x` where `fun(x) = 0`. This solution is where `fun(x)` changes sign—`fzero` cannot find a root of a function such as `x^2`.

`x = fzero(fun,x0,options)` uses `options` to modify the solution process.

`x = fzero(problem)` solves a root-finding problem specified by `problem`.

`[x,fval,exitflag,output] = fzero( ___ )` returns `fun(x)` in the `fval` output, `exitflag` encoding the reason `fzero` stopped, and an output structure containing information on the solution process.

Input Arguments

fun - Function to solve

function handle

Function to solve, specified as a handle to a scalar-valued function. `fun` accepts a scalar `x` and returns a scalar `fun(x)`.

`fzero` solves `fun(x) = 0`. To solve an equation `fun(x) = c(x)`, instead solve `fun2(x) = fun(x) - c(x) = 0`.

To include extra parameters in your function, see the example "Root of Function with Extra Parameter" on page 1-1895 and the section "Parameterizing Functions".

Example: `@sin`

Example: `@myFunction`

Example: `@(x)(x-a)^5 - 3*x + a - 1`

Data Types
`function_handle`

x0 - Initial value

scalar | 2-element vector

Initial value, specified as a real scalar or a 2-element real vector.

- Scalar — `fzero` begins at `x0` and tries to locate a point `x1` where `fun(x1)` has the opposite sign of `fun(x0)`. Then `fzero` iteratively shrinks the interval where `fun` changes sign to reach a solution.
- 2-element vector — `fzero` checks that `fun(x0(1))` and `fun(x0(2))` have opposite signs, and errors if they do not. It then iteratively shrinks the interval where `fun` changes sign to reach a solution. An interval `x0` must be finite; it cannot contain ±`Inf`.

Tip Calling `fzero` with an interval (`x0` with two elements) is often faster than calling it with a scalar `x0`.

Example: 3

Example: [2,17]

Data Types
`double`

options - Options for solution process

Options for solution process, specified as a structure. Create or modify the `options` structure using `optimset`. `fzero` uses these `options` structure fields.

`Display`	Level of display: • `'off'` displays no output. • `'iter'` displays output at each iteration. • `'final'` displays just the final output. • `'notify'` (default) displays output only if the function does not converge.
`FunValCheck`	Check whether objective function values are valid. • `'on'` displays an error when the objective function returns a value that is `complex`, `Inf`, or `NaN`. • The default, `'off'`, displays no error.
`OutputFcn`	Specify one or more user-defined functions that an optimization function calls at each iteration, either as a function handle or as a cell array of function handles. The default is none (`[]`). See "Output Functions".
`PlotFcns`	Plot various measures of progress while the algorithm executes. Select from predefined plots or write your own. Pass a function handle or a cell array of function handles. The default is none (`[]`). • `@optimplotx` plots the current point. • `@optimplotfval` plots the function value. For information on writing a custom plot function, see "Plot Functions".
`TolX`	Termination tolerance on `x`, a positive scalar. The default is `eps`, 2.2204e–16.

Example: `options = optimset('FunValCheck','on')`

Data Types
`struct`

problem - Root-finding problem

Root-finding problem, specified as a structure with all of the following fields.

`objective`	Objective function
`x0`	Initial point for `x`, real scalar or 2-element vector
`solver`	`'fzero'`
`options`	Options structure created with `optimset`

For an example, see "Solve Problem Structure" on page 1-1897.

Data Types
`struct`

Output Arguments

x - Location of root or sign change

Real scalar

Location of root or sign change, returned as a scalar.

fval - Function value at `x`

Real scalar

Function value at `x`, returned as a scalar.

exitflag - Integer encoding the exit condition

Integer encoding the exit condition, meaning the reason `fsolve` stopped its iterations.

`1`	Function converged to a solution `x`.
`-1`	Algorithm was terminated by the output function or plot function.
`-3`	`NaN` or `Inf` function value was encountered while searching for an interval containing a sign change.

`-4`	Complex function value was encountered while searching for an interval containing a sign change.
`-5`	Algorithm might have converged to a singular point.
`-6`	`fzero` did not detect a sign change.

output - Information about root-finding process

Information about root-finding process, returned as a structure. The fields of the structure are:

`intervaliterations`	Number of iterations taken to find an interval containing a root
`iterations`	Number of zero-finding iterations
`funcCount`	Number of function evaluations
`algorithm`	`'bisection, interpolation'`
`message`	Exit message

Examples

Root Starting From One Point

Calculate π by finding the zero of the sine function near 3.

```
fun = @sin; % function
x0 = 3; % initial point
x = fzero(fun,x0)
```

```
x =
    3.1416
```

Root Starting From an Interval

Find the zero of cosine between 1 and 2.

```
fun = @cos; % function
x = [1 2]; % initial interval
x = fzero(fun,x0)
```

```
x =
   1.5708
```

Note that `cos(1)` and `cos(2)` differ in sign.

Root of a Function Defined by a File

Find a zero of the function

$$f(x) = x^3 - 2x - 5.$$

First, write a file called `f.m`.

```
function y = f(x)
y = x.^3-2*x-5;
```

Save `f.m` on your MATLAB path.

Find the zero of $f(x)$ near `2`.

```
fun = @f; % function
x0 = 2; % initial point
z = fzero(fun,x0)

z =
    2.0946
```

Since `f(x)` is a polynomial, you can find the same real zero, and a complex conjugate pair of zeros, using the `roots` command.

```
roots([1 0 -2 -5])

   ans =
   2.0946
  -1.0473 + 1.1359i
  -1.0473 - 1.1359i
```

Root of Function with Extra Parameter

Find the root of a function that has an extra parameter.

```
myfun = @(x,c) cos(c*x);  % parameterized function
c = 2;                    % parameter
fun = @(x) myfun(x,c);    % function of x alone
x = fzero(fun,0.1)

x =

    0.7854
```

Nondefault Options

Plot the solution process by setting some plot functions.

Define the function and initial point.

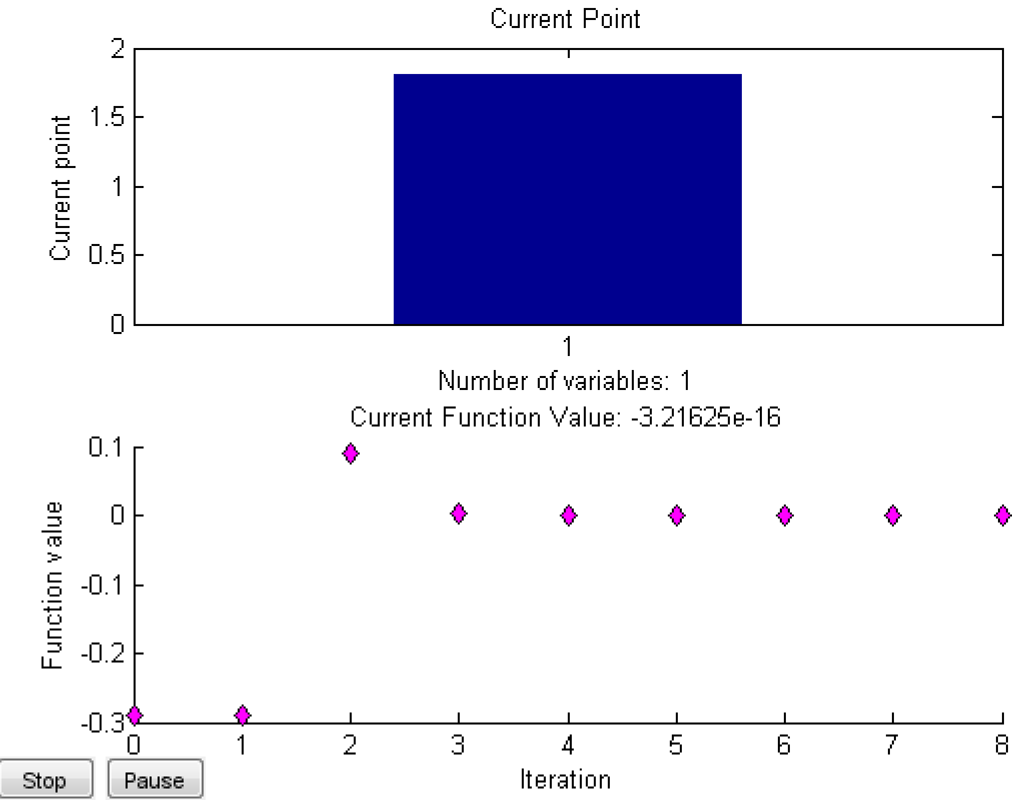
```
fun = @(x)sin(cosh(x));
x0 = 1;
```

Examine the solution process by setting options that include plot functions.

```
options = optimset('PlotFcns',{@optimplotx,@optimplotfval});
```

Run `fzero` including the options.

```
x = fzero(fun,x0,options)

x =

    1.8115
```



Solve Problem Structure

Solve a problem that is defined by a problem structure.

Define a structure that encodes a root-finding problem.

```
problem.objective = @(x)sin(cosh(x));
problem.x0 = 1;
problem.solver = 'fzero'; % a required part of the structure
problem.options = optimset(@fzero); % default options
```

Solve the problem.

```
x = fzero(problem)

x =

    1.8115
```

More Information from Solution

Find the point where `exp(-exp(-x)) = x`, and display information about the solution process.

```
fun = @(x) exp(-exp(-x)) - x; % function
x0 = [0 1]; % initial interval
options = optimset('Display','iter'); % show iterations
[x fval exitflag output] = fzero(fun,x0,options)
```

```
 Func-count    x          f(x)             Procedure
    2               1     -0.307799        initial
    3        0.544459     0.0153522        interpolation
    4        0.566101    0.00070708        interpolation
    5        0.567143  -1.40255e-08        interpolation
    6        0.567143   1.50013e-12        interpolation
    7        0.567143             0        interpolation

Zero found in the interval [0, 1]
x =
    0.5671
fval =
     0
exitflag =
     1
output =
    intervaliterations: 0
            iterations: 5
             funcCount: 7
             algorithm: 'bisection, interpolation'
               message: 'Zero found in the interval [0, 1]'
```

`fval` = 0 means `fun(x) = 0`, as desired.

Algorithms

The `fzero` command is a function file. The algorithm, created by T. Dekker, uses a combination of bisection, secant, and inverse quadratic interpolation methods. An Algol 60 version, with some improvements, is given in [1]. A Fortran version, upon which `fzero` is based, is in [2].

References

[1] Brent, R., *Algorithms for Minimization Without Derivatives*, Prentice-Hall, 1973.

[2] Forsythe, G. E., M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, 1976.

See Also

`fminbnd | optimset | roots`

Related Examples

- “Roots of Scalar Functions”
- “Parameterizing Functions”

gallery

Purpose

Test matrices

Syntax

```
[A,B,C,...] = gallery(matname,P1,P2,...)
[A,B,C,...] = gallery(matname,P1,P2,...,classname)
gallery(3)
gallery(5)
```

Description

`[A,B,C,...] = gallery(matname,P1,P2,...)` returns the test matrices specified by the quoted string `matname`. The `matname` input is the name of a matrix family selected from the table below. `P1,P2,...` are input parameters required by the individual matrix family. The number of optional parameters `P1,P2,...` used in the calling syntax varies from matrix to matrix. The exact calling syntaxes are detailed in the individual matrix descriptions below.

`[A,B,C,...] = gallery(matname,P1,P2,...,classname)` produces a matrix of class `classname`. The `classname` input is a quoted string that must be either `'single'` or `'double'` (unless `matname` is `'integerdata'`, in which case `'int8'`, `'int16'`, `'int32'`, `'uint8'`, `'uint16'`, and `'uint32'` are also allowed). If `classname` is not specified, then the class of the matrix is determined from those arguments among `P1,P2,...` that do not specify dimensions or select an option. If any of these arguments is of class `single` then the matrix is `single`; otherwise the matrix is `double`.

`gallery(3)` is a badly conditioned 3-by-3 matrix and `gallery(5)` is an interesting eigenvalue problem.

The gallery holds over fifty different test matrix functions useful for testing algorithms and other purposes.

- `binomial`
- `cauchy`
- `chebspec`
- `chebvand`
- `chow`

- `circul`
- `clement`
- `compar`
- `condex`
- `cycol`
- `dorr`
- `dramadah`
- `fiedler`
- `forsythe`
- `frank`
- `gearmat`
- `gcdmat`
- `grcar`
- `hanowa`
- `house`
- `integerdata`
- `invhess`
- `invol`
- `ipjfact`
- `jordbloc`
- `kahan`
- `kms`
- `krylov`
- `lauchli`
- `lehmer`

- leslie
- lesp
- lotkin
- minij
- moler
- neumann
- normaldata
- orthog
- parter
- pei
- poisson
- prolate
- randcolu
- randcorr
- randhess
- randjorth
- rando
- randsvd
- redheff
- riemann
- ris
- sampling
- smoke
- toeppd
- tridiag

- `triw`
- `uniformdata`
- `wathen`
- `wilk`

binomial — Multiple of involutory matrix

`A = gallery('binomial',n)` returns an `n`-by-`n` matrix,with integer entries such that `A^2 = 2^(n-1)*eye(n)`.

Thus, `B = A*2^((1-n)/2)` is involutory, that is, `B^2 = eye(n)`.

cauchy — Cauchy matrix

`C = gallery('cauchy',x,y)` returns an `n`-by-`n` matrix, `C(i,j) = 1/(x(i)+y(j))`. Arguments `x` and `y` are vectors of length `n`. If you pass in scalars for `x` and `y`, they are interpreted as vectors `1:x` and `1:y`.

`C = gallery('cauchy',x)` returns the same as above with `y = x`. That is, the command returns `C(i,j) = 1/(x(i)+x(j))`.

Explicit formulas are known for the inverse and determinant of a Cauchy matrix. The determinant `det(C)` is nonzero if `x` and `y` both have distinct elements. `C` is totally positive if `0 < x(1) <...  < x(n)` and `0 < y(1) < ...  < y(n)`.

chebspec — Chebyshev spectral differentiation matrix

`C = gallery('chebspec',n,switch)` returns a Chebyshev spectral differentiation matrix of order `n`. Argument `switch` is a variable that determines the character of the output matrix. By default, `switch` = `0`.

For `switch` = `0` (“no boundary conditions”), `C` is nilpotent (`C^n = 0`) and has the null vector `ones(n,1)`. The matrix `C` is similar to a Jordan block of size `n` with eigenvalue zero.

For `switch` = `1`, `C` is nonsingular and well-conditioned, and its eigenvalues have negative real parts.

The eigenvector matrix of the Chebyshev spectral differentiation matrix is ill-conditioned.

chebvand — Vandermonde-like matrix for the Chebyshev polynomials

`C = gallery('chebvand',p)` produces the (primal) Chebyshev Vandermonde matrix based on the vector of points `p`, which define where the Chebyshev polynomial is calculated.

`C = gallery('chebvand',m,p)` where `m` is scalar, produces a rectangular version of the above, with `m` rows.

If `p` is a vector, then $C(i,j) = T_{i-1}(p(j))$ where T_{i-1} is the Chebyshev polynomial of degree $i - 1$. If `p` is a scalar, then `p` equally spaced points on the interval `[0,1]` are used to calculate `C`.

chow — Singular Toeplitz lower Hessenberg matrix

`A = gallery('chow',n,alpha,delta)` returns `A` such that `A = H(alpha) + delta*eye(n)`, where $H_{i,j}(\alpha) = \alpha^{(i-j+1)}$ and argument `n` is the order of the Chow matrix. Default value for scalars `alpha` and `delta` are `1` and `0`, respectively.

`H(alpha)` has `p = floor(n/2)` eigenvalues that are equal to zero. The rest of the eigenvalues are equal to `4*alpha*cos(k*pi/(n+2))^2, k=1:n-p`.

circul — Circulant matrix

`C = gallery('circul',v)` returns the circulant matrix whose first row is the vector `v`.

A circulant matrix has the property that each row is obtained from the previous one by cyclically permuting the entries one step forward. It is a special Toeplitz matrix in which the diagonals “wrap around.”

If `v` is a scalar, then `C = gallery('circul',1:v)`.

The eigensystem of `C` (`n`-by-`n`) is known explicitly: If `t` is an `n`th root of unity, then the inner product of `v` and $w = [1\ t\ t^2\ \ldots\ t^{(n-1)}]$ is an eigenvalue of `C` and `w(n:-1:1)` is an eigenvector.

clement — Tridiagonal matrix with zero diagonal entries

`A = gallery('clement',n,k)` returns an `n`-by-`n` tridiagonal matrix with zeros on its main diagonal and known eigenvalues. It is singular if `n` is odd. About 64 percent of the entries of the inverse are zero. The eigenvalues include plus and minus the numbers `n-1`, `n-3`, `n-5`, `...`, (1 or 0).

For `k=0` (the default), `A` is nonsymmetric. For `k=1`, `A` is symmetric.

`gallery('clement',n,1)` is diagonally similar to `gallery('clement',n)`.

For odd `N = 2*M+1, M+1` of the singular values are the integers `sqrt((2*M+1)^2 - (2*K+1).^2), K = 0:M`.

Note Similar properties hold for `gallery('tridiag',x,y,z)` where `y = zeros(n,1)`. The eigenvalues still come in plus/minus pairs but they are not known explicitly.

---

compar — Comparison matrices

`A = gallery('compar',A,1)` returns `A` with each diagonal element replaced by its absolute value, and each off-diagonal element replaced by minus the absolute value of the largest element in absolute value in its row. However, if `A` is triangular `compar(A,1)` is too.

`gallery('compar',A)` is `diag(B) - tril(B,-1) - triu(B,1)`, where `B = abs(A)`. `compar(A)` is often denoted by M(*A*) in the literature.

`gallery('compar',A,0)` is the same as `gallery('compar',A)`.

condex — Counter-examples to matrix condition number estimators

`A = gallery('condex',n,k,theta)` returns a “counter-example” matrix to a condition estimator. It has order `n` and scalar parameter `theta` (default 100).

The matrix, its natural size, and the estimator to which it applies are specified by `k`:

`k = 1`	4-by-4	LINPACK
`k = 2`	3-by-3	LINPACK
`k = 3`	arbitrary	LINPACK (`rcond`) (independent of `theta`)
`k = 4`	`n >= 4`	LAPACK (RCOND) (default). It is the inverse of this matrix that is a counter-example.

If `n` is not equal to the natural size of the matrix, then the matrix is padded out with an identity matrix to order `n`.

cycol — Matrix whose columns repeat cyclically

`A = gallery('cycol',[m n],k)` returns an `m`-by-`n` matrix with cyclically repeating columns, where one "cycle" consists of `randn(m,k)`. Thus, the rank of matrix `A` cannot exceed `k`, and `k` must be a scalar.

Argument `k` defaults to `round(n/4)`, and need not evenly divide `n`.

`A = gallery('cycol',n,k)`, where `n` is a scalar, is the same as `gallery('cycol',[n n],k)`.

dorr — Diagonally dominant, ill-conditioned, tridiagonal matrix

`[c,d,e] = gallery('dorr',n,theta)` returns the vectors defining an `n`-by-`n`, row diagonally dominant, tridiagonal matrix that is ill-conditioned for small nonnegative values of `theta`. The default value of `theta` is `0.01`. The Dorr matrix itself is the same as `gallery('tridiag',c,d,e)`.

`A = gallery('dorr',n,theta)` returns the matrix itself, rather than the defining vectors.

dramadah — Matrix of zeros and ones whose inverse has large integer entries

`A = gallery('dramadah',n,k)` returns an `n`-by-`n` matrix of `0`'s and `1`'s for which `mu(A) = norm(inv(A),'fro')` is relatively large, although not necessarily maximal. An anti-Hadamard matrix `A` is a matrix with elements `0` or `1` for which `mu(A)` is maximal.

`n` and `k` must both be scalars. Argument `k` determines the character of the output matrix:

k = 1	Default. `A` is Toeplitz, with `abs(det(A)) = 1`, and `mu(A) > c(1.75)^n`, where `c` is a constant. The inverse of `A` has integer entries.
k = 2	`A` is upper triangular and Toeplitz. The inverse of `A` has integer entries.
k = 3	`A` has maximal determinant among lower Hessenberg (0,1) matrices. `det(A)` = the `n`th Fibonacci number. `A` is Toeplitz. The eigenvalues have an interesting distribution in the complex plane.

fiedler — Symmetric matrix

`A = gallery('fiedler',c)`, where `c` is a length `n` vector, returns the `n`-by-`n` symmetric matrix with elements `abs(n(i)-n(j))`. For scalar `c`, `A = gallery('fiedler',1:c)`.

Matrix `A` has a dominant positive eigenvalue and all the other eigenvalues are negative.

Explicit formulas for `inv(A)` and `det(A)` are given in [Todd, J., *Basic Numerical Mathematics*, Vol. 2: Numerical Algebra, Birkhauser, Basel, and Academic Press, New York, 1977, p. 159] and attributed to Fiedler. These indicate that `inv(A)` is tridiagonal except for nonzero `(1,n)` and `(n,1)` elements.

forsythe — Perturbed Jordan block

`A = gallery('forsythe',n,alpha,lambda)` returns the `n`-by-`n` matrix equal to the Jordan block with eigenvalue `lambda`, excepting that

A(n,1) = alpha. The default values of scalars alpha and lambda are sqrt(eps) and 0, respectively.

The characteristic polynomial of A is given by:

```
det(A-t*I) = (lambda-t)^N - alpha*(-1)^n.
```

frank — Matrix with ill-conditioned eigenvalues

F = gallery('frank',n,k) returns the Frank matrix of order n. It is upper Hessenberg with determinant 1. If k = 1, the elements are reflected about the anti-diagonal (1,n) — (n,1). The eigenvalues of F may be obtained in terms of the zeros of the Hermite polynomials. They are positive and occur in reciprocal pairs; thus if n is odd, 1 is an eigenvalue. F has floor(n/2) ill-conditioned eigenvalues — the smaller ones.

gcdmat — Greatest common divisor matrix

A = gallery('gcdmat',n) returns the n-by-n matrix with (i,j) entry gcd(i,j). MatrixA is symmetric positive definite, and A.^r is symmetric positive semidefinite for all nonnegative r.

gearmat — Gear matrix

A = gallery('gearmat',n,i,j) returns the n-by-n matrix with ones on the sub- and super-diagonals, sign(i) in the (1,abs(i)) position, sign(j) in the (n,n+1-abs(j)) position, and zeros everywhere else. Arguments i and j default to n and -n, respectively.

Matrix A is singular, can have double and triple eigenvalues, and can be defective.

All eigenvalues are of the form 2*cos(a) and the eigenvectors are of the form [sin(w+a), sin(w+2*a), ..., sin(w+n*a)], where a and w are given in Gear, C. W., "A Simple Set of Test Matrices for Eigenvalue Programs," *Math. Comp.*, Vol. 23 (1969), pp. 119-125.

grcar — Toeplitz matrix with sensitive eigenvalues

A = gallery('grcar',n,k) returns an n-by-n Toeplitz matrix with -1s on the subdiagonal, 1s on the diagonal, and k superdiagonals of 1s. The default is k = 3. The eigenvalues are sensitive.

hanowa — Matrix whose eigenvalues lie on a vertical line in the complex plane

`A = gallery('hanowa',n,d)` returns an `n`-by-`n` block `2`-by-`2` matrix of the form:

```
[d*eye(m) -diag(1:m)
diag(1:m) d*eye(m)]
```

Argument `n` is an even integer `n=2*m`. Matrix `A` has complex eigenvalues of the form `d` ± `k*i`, for `1 <= k <= m`. The default value of `d` is `-1`.

house — Householder matrix

`[v,beta,s] = gallery('house',x,k)` takes `x`, an `n`-element column vector, and returns `V` and `beta` such that `H*x = s*e1`. In this expression, `e1` is the first column of `eye(n)`, `abs(s) = norm(x)`, and `H = eye(n) - beta*V*V'` is a Householder matrix.

`k` determines the sign of `s`:

`k = 0`	`sign(s) = -sign(x(1))` (default)
`k = 1`	`sign(s) = sign(x(1))`
`k = 2`	`sign(s) = 1` (`x` must be real)

If `x` is complex, then `sign(x) = x./abs(x)` when `x` is nonzero.

If `x = 0`, or if `x = alpha*e1` (`alpha >= 0`) and either `k = 1` or `k = 2`, then `V = 0`, `beta = 1`, and `s = x(1)`. In this case, `H` is the identity matrix, which is not strictly a Householder matrix.

`[v, beta] = gallery('house',x)` takes `x`, a scalar or `n`-element column vector, and returns `v` and `beta` such that `eye(n,n) - beta*v*v'` is a Householder matrix. A Householder matrix `H` satisfies the relationship

```
H*x = -sign(x(1))*norm(x)*e1
```

where `e1` is the first column of `eye(n,n)`. Note that if `x` is complex, then `sign(x) exp(i*arg(x))` (which equals `x./abs(x)` when `x` is nonzero).

If `x = 0`, then `v = 0` and `beta = 1`.

integerdata — Array of arbitrary data from uniform distribution on specified range of integers

`A = gallery('integerdata',imax,[m,n,...],j)` returns an `m`-by-`n`-by-... array `A` whose values are a sample from the uniform distribution on the integers `1:imax`. `j` must be an integer value in the interval `[0, 2^32-1]`. Calling `gallery('integerdata', ...)` with different values of `J` will return different arrays. Repeated calls to `gallery('integerdata',...)` with the same `imax`, size vector and `j` inputs will always return the same array.

In any call to `gallery('integerdata', ...)` you can substitute individual inputs `m`,`n`,... for the size vector input `[m,n,...]`. For example, `gallery('integerdata',7,[1,2,3,4],5)` is equivalent to `gallery('integerdata',7,1,2,3,4,5)`.

`A = gallery('integerdata',[imin imax],[m,n,...],j)` returns an `m`-by-`n`-by-... array `A` whose values are a sample from the uniform distribution on the integers `imin:imax`.

`[A,B,...]  = gallery('integerdata',[imin imax],[m,n,...],j)` returns multiple `m`-by-`n`-by-... arrays `A`, `B`, ..., containing different values.

`A = gallery('integerdata',[imin imax],[m,n,...],j,classname)` produces an array of class `classname`. `classname` must be `'uint8'`, `'uint16'`, `'uint32'`, `'int8'`, `'int16'`, `int32'`, `'single'` or `'double'`.

invhess — Inverse of an upper Hessenberg matrix

`A = gallery('invhess',x,y)`, where `x` is a length `n` vector and `y` is a length `n-1` vector, returns the matrix whose lower triangle agrees with that of `ones(n,1)*x'` and whose strict upper triangle agrees with that of `[1 y]*ones(1,n)`.

The matrix is nonsingular if `x(1) ~= 0` and `x(i+1) ~= y(i)` for all `i`, and its inverse is an upper Hessenberg matrix. Argument `y` defaults to `-x(1:n-1)`.

If `x` is a scalar, `invhess(x)` is the same as `invhess(1:x)`.

invol — Involutory matrix

`A = gallery('invol',n)` returns an `n`-by-`n` involutory (`A*A = eye(n)`) and ill-conditioned matrix. It is a diagonally scaled version of `hilb(n)`.

`B = (eye(n)-A)/2` and `B = (eye(n)+A)/2` are idempotent (`B*B = B`).

ipjfact — Hankel matrix with factorial elements

`[A,d] = gallery('ipjfact',n,k)` returns `A`, an `n`-by-`n` Hankel matrix, and `d`, the determinant of `A`, which is known explicitly. If `k = 0` (the default), then the elements of `A` are `A(i,j) = (i+j)!` If `k = 1`, then the elements of `A` are `A(i,j) 1/(i+j)`.

Note that the inverse of `A` is also known explicitly.

jordbloc — Jordan block

`A = gallery('jordbloc',n,lambda)` returns the `n`-by-`n` Jordan block with eigenvalue `lambda`. The default value for `lambda` is `1`.

kahan — Upper trapezoidal matrix

`A = gallery('kahan',n,theta,pert)` returns an upper trapezoidal matrix that has interesting properties regarding estimation of condition and rank.

If `n` is a two-element vector, then `A` is `n(1)`-by-`n(2)`; otherwise, `A` is `n`-by-`n`. The useful range of `theta` is `0 < theta < pi`, with a default value of `1.2`.

To ensure that the QR factorization with column pivoting does not interchange columns in the presence of rounding errors, the diagonal is perturbed by `pert*eps*diag([n:-1:1])`. The default `pert` is `25`, which ensures no interchanges for `gallery('kahan',n)` up to at least `n = 90` in IEEE arithmetic.

kms — Kac-Murdock-Szego Toeplitz matrix

`A = gallery('kms',n,rho)` returns the `n`-by-`n` Kac-Murdock-Szego Toeplitz matrix such that `A(i,j) = rho^(abs(i-j))`, for real `rho`.

For complex `rho`, the same formula holds except that elements below the diagonal are conjugated. `rho` defaults to 0.5.

The KMS matrix `A` has these properties:

- An LDL' factorization with `L inv(gallery('triw',n,-rho,1))'`, and `D(i,i) (1-abs(rho)^2)*eye(n)`, except `D(1,1) = 1`.
- Positive definite if and only if `0 < abs(rho) < 1`.
- The inverse `inv(A)` is tridiagonal.

krylov — Krylov matrix

`B = gallery('krylov',A,x,j)` returns the Krylov matrix

```
[x, Ax, A^2x, ..., A^(j-1)x]
```

where `A` is an `n`-by-`n` matrix and `x` is a length `n` vector. The defaults are `x ones(n,1)`, and `j = n`.

`B = gallery('krylov',n)` is the same as `gallery('krylov',(randn(n))`.

lauchli — Rectangular matrix

`A = gallery('lauchli',n,mu)` returns the `(n+1)`-by-`n` matrix

```
[ones(1,n); mu*eye(n)]
```

The Lauchli matrix is a well-known example in least squares and other problems that indicates the dangers of forming `A'*A`. Argument `mu` defaults to `sqrt(eps)`.

lehmer — Symmetric positive definite matrix

`A = gallery('lehmer',n)` returns the symmetric positive definite `n`-by-`n` matrix such that `A(i,j) = i/j` for `j >= i`.

The Lehmer matrix `A` has these properties:

- A is totally nonnegative.
- The inverse inv(A) is tridiagonal and explicitly known.
- The order n <= cond(A) <= 4*n*n.

leslie — Matrix of birth numbers and survival rates

L = gallery('leslie',a,b) is the n-by-n matrix from the Leslie population model with average birth numbers a(1:n) and survival rates b(1:n-1). It is zero, apart from the first row (which contains the a(i)) and the first subdiagonal (which contains the b(i)). For a valid model, the a(i) are nonnegative and the b(i) are positive and bounded by 1, i.e., 0 < b(i) <= 1.

L = gallery('leslie',n) generates the Leslie matrix with a = ones(n,1), b = ones(n-1,1).

lesp — Tridiagonal matrix with real, sensitive eigenvalues

A = gallery('lesp',n) returns an n-by-n matrix whose eigenvalues are real and smoothly distributed in the interval approximately [-2*N-3.5, -4.5].

The sensitivities of the eigenvalues increase exponentially as the eigenvalues grow more negative. The matrix is similar to the symmetric tridiagonal matrix with the same diagonal entries and with off-diagonal entries 1, via a similarity transformation with D = diag(1!,2!,...,n!).

lotkin — Lotkin matrix

A = gallery('lotkin',n) returns the Hilbert matrix with its first row altered to all ones. The Lotkin matrix A is nonsymmetric, ill-conditioned, and has many negative eigenvalues of small magnitude. Its inverse has integer entries and is known explicitly.

minij — Symmetric positive definite matrix

A = gallery('minij',n) returns the n-by-n symmetric positive definite matrix with A(i,j) = min(i,j).

The minij matrix has these properties:

- The inverse `inv(A)` is tridiagonal and equal to `-1` times the second difference matrix, except its `(n,n)` element is `1`.
- Givens' matrix, `2*A-ones(size(A))`, has tridiagonal inverse and eigenvalues `0.5*sec((2*r-1)*pi/(4*n))^2`, where `r=1:n`.
- `(n+1)*ones(size(A))-A` has elements that are `max(i,j)` and a tridiagonal inverse.

moler — Symmetric positive definite matrix

`A = gallery('moler',n,alpha)` returns the symmetric positive definite `n`-by-`n` matrix `U'*U`, where `U = gallery('triw',n,alpha)`.

For the default `alpha` = `-1`, `A(i,j) = min(i,j)-2`, and `A(i,i) = i`. One of the eigenvalues of `A` is small.

neumann — Singular matrix from the discrete Neumann problem (sparse)

`C = gallery('neumann',n)` returns the sparse `n`-by-`n` singular, row diagonally dominant matrix resulting from discretizing the Neumann problem with the usual five-point operator on a regular mesh. Argument `n` is a perfect square integer $n = m^2$ or a two-element vector. `C` is sparse and has a one-dimensional null space with null vector `ones(n,1)`.

normaldata — Array of arbitrary data from standard normal distribution

`A = gallery('normaldata',[m,n,...],j)` returns an `m`-by-`n`-by-... array `A`. The values of `A` are a random sample from the standard normal distribution. `j` must be an integer value in the interval `[0, 2^32-1]`. Calling `gallery('normaldata', ...)` with different values of `j` will return different arrays. Repeated calls to `gallery('normaldata',...)` with the same size vector and `j` inputs will always return the same array.

In any call to `gallery('normaldata', ...)` you can substitute individual inputs `m`,`n`,... for the size vector input `[m,n,...]`. For example, `gallery('normaldata',[1,2,3,4],5)` is equivalent to `gallery('normaldata',1,2,3,4,5)`.

`[A,B,...] = gallery('normaldata',[m,n,...],j)` returns multiple `m`-by-`n`-by-... arrays `A`, `B`, ..., containing different values.

`A = gallery('normaldata',[m,n,...],j, classname)` produces a matrix of class `classname`. `classname` must be either `'single'` or `'double'`.

Generate the arbitrary 6-by-4 matrix of data from the standard normal distribution `N(0, 1)` corresponding to `j = 2`:.

```
x = gallery('normaldata', [6, 4], 2);
```

Generate the arbitrary 1-by-2-by-3 single array of data from the standard normal distribution `N(0, 1)` corresponding to `j = 17`:.

```
y = gallery('normaldata', 1, 2, 3, 17, 'single');
```

orthog — Orthogonal and nearly orthogonal matrices

`Q = gallery('orthog',n,k)` returns the kth type of matrix of order `n`, where `k > 0` selects exactly orthogonal matrices, and `k < 0` selects diagonal scalings of orthogonal matrices. Available types are:

`k = 1`	`Q(i,j) = sqrt(2/(n+1)) * sin(i*j*pi/(n+1))` Symmetric eigenvector matrix for second difference matrix. This is the default.
`k = 2`	`Q(i,j) = 2/(sqrt(2*n+1)) * sin(2*i*j*pi/(2*n+1))` Symmetric.
`k = 3`	`Q(r,s) = exp(2*pi*i*(r-1)*(s-1)/n) / sqrt(n)` Unitary, the Fourier matrix. Q^4 is the identity. This is essentially the same matrix as `fft(eye(n))/sqrt(n)`!
`k = 4`	Helmert matrix: a permutation of a lower Hessenberg matrix, whose first row is `ones(1:n)/sqrt(n)`.

`k = 5`	`Q(i,j) = sin(2*pi*(i-1)*(j-1)/n) + cos(2*pi*(i-1)*(j-1)/n)` Symmetric matrix arising in the Hartley transform.
`k = 6`	`Q(i,j) = sqrt(2/n)*cos((i-1/2)*(j-1/2)*pi/n)` Symmetric matrix arising as a discrete cosine transform.
`k = -1`	`Q(i,j) = cos((i-1)*(j-1)*pi/(n-1))` Chebyshev Vandermonde-like matrix, based on extrema of `T(n-1)`.
`k = -2`	`Q(i,j) = cos((i-1)*(j-1/2)*pi/n))` Chebyshev Vandermonde-like matrix, based on zeros of `T(n)`.

parter — Toeplitz matrix with singular values near pi

`C = gallery('parter',n)` returns the matrix `C` such that `C(i,j) = 1/(i-j+0.5)`.

`C` is a Cauchy matrix and a Toeplitz matrix. Most of the singular values of `C` are very close to `pi`.

pei — Pei matrix

`A = gallery('pei',n,alpha)`, where `alpha` is a scalar, returns the symmetric matrix `alpha*eye(n) + ones(n)`. The default for `alpha` is 1. The matrix is singular for `alpha` equal to either `0` or `-n`.

poisson — Block tridiagonal matrix from Poisson's equation (sparse)

`A = gallery('poisson',n)` returns the block tridiagonal (sparse) matrix of order `n^2` resulting from discretizing Poisson's equation with the 5-point operator on an `n`-by-`n` mesh.

prolate — Symmetric, ill-conditioned Toeplitz matrix

`A = gallery('prolate',n,w)` returns the `n`-by-`n` prolate matrix with parameter `w`. It is a symmetric Toeplitz matrix.

If `0 < w < 0.5` then A is positive definite

- The eigenvalues of A are distinct, lie in `(0,1)`, and tend to cluster around 0 and 1.
- The default value of w is 0.25.

randcolu — Random matrix with normalized cols and specified singular values

`A = gallery('randcolu',n)` is a random n-by-n matrix with columns of unit 2-norm, with random singular values whose squares are from a uniform distribution.

`A'*A` is a correlation matrix of the form produced by `gallery('randcorr',n)`.

`gallery('randcolu',x)` where x is an n-vector (n > 1), produces a random n-by-n matrix having singular values given by the vector x. The vector x must have nonnegative elements whose sum of squares is n.

`gallery('randcolu',x,m)` where `m >= n`, produces an m-by-n matrix.

`gallery('randcolu',x,m,k)` provides a further option:

`k = 0`	`diag(x)` is initially subjected to a random two-sided orthogonal transformation, and then a sequence of Givens rotations is applied (default).
`k = 1`	The initial transformation is omitted. This is much faster, but the resulting matrix may have zero entries.

For more information, see:

References

[1] Davies, P. I. and N. J. Higham, "Numerically Stable Generation of Correlation Matrices and Their Factors," *BIT*, Vol. 40, 2000, pp. 640-651.

randcorr — Random correlation matrix with specified eigenvalues

`gallery('randcorr',n)` is a random `n`-by-`n` correlation matrix with random eigenvalues from a uniform distribution. A correlation matrix is a symmetric positive semidefinite matrix with `1`s on the diagonal (see `corrcoef`).

`gallery('randcorr',x)` produces a random correlation matrix having eigenvalues given by the vector `x`, where `length(x) > 1`. The vector `x` must have nonnegative elements summing to `length(x)`.

`gallery('randcorr',x,k)` provides a further option:

`k = 0`	The diagonal matrix of eigenvalues is initially subjected to a random orthogonal similarity transformation, and then a sequence of Givens rotations is applied (default).
`k = 1`	The initial transformation is omitted. This is much faster, but the resulting matrix may have some zero entries.

For more information, see:

References

[1] Bendel, R. B. and M. R. Mickey, "Population Correlation Matrices for Sampling Experiments," *Commun. Statist. Simulation Comput.*, B7, 1978, pp. 163-182.

[2] Davies, P. I. and N. J. Higham, "Numerically Stable Generation of Correlation Matrices and Their Factors," *BIT*, Vol. 40, 2000, pp. 640-651.

randhess — Random, orthogonal upper Hessenberg matrix

`H = gallery('randhess',n)` returns an `n`-by-`n` real, random, orthogonal upper Hessenberg matrix.

`H = gallery('randhess',x)` if `x` is an arbitrary, real, length `n` vector with `n > 1`, constructs `H` nonrandomly using the elements of `x` as parameters.

Matrix `H` is constructed via a product of `n-1` Givens rotations.

randjorth — Random J-orthogonal matrix

`A = gallery('randjorth', n)`, for a positive integer `n`, produces a random `n`-by-`n` `J`-orthogonal matrix `A`, where

- `J = blkdiag(eye(ceil(n/2)),-eye(floor(n/2)))`
- `cond(A) = sqrt(1/eps)`

`J`-orthogonality means that A'*J*A = J. Such matrices are sometimes called *hyperbolic*.

`A = gallery('randjorth', n, m)`, for positive integers `n` and `m`, produces a random (`n+m`)-by-(`n+m`) `J`-orthogonal matrix `A`, where

- `J = blkdiag(eye(n),-eye(m))`
- `cond(A) = sqrt(1/eps)`

```
A = gallery('randjorth',n,m,c,symm,method)
```

uses the following optional input arguments:

- `c` — Specifies `cond(A)` to be the scalar `c`.
- `symm` — Enforces symmetry if the scalar `symm` is nonzero.
- `method` — calls `qr` to perform the underlying orthogonal transformations if the scalar `method` is nonzero. A call to `qr` is much faster than the default method for large dimensions

rando — Random matrix composed of elements -1, 0 or 1

`A = gallery('rando',n,k)` returns a random `n`-by-`n` matrix with elements from one of the following discrete distributions:

`k = 1`	`A(i,j)` = `0` or `1` with equal probability (default).
`k = 2`	`A(i,j)` = `-1` or `1` with equal probability.
`k = 3`	`A(i,j)` = `-1`, `0` or `1` with equal probability.

Argument `n` may be a two-element vector, in which case the matrix is `n(1)`-by-`n(2)`.

randsvd — Random matrix with preassigned singular values

`A = gallery('randsvd',n,kappa,mode,kl,ku)` returns a banded (multidiagonal) random matrix of order `n` with `cond(A) = kappa` and singular values from the distribution mode. If `n` is a two-element vector, `A` is `n(1)`-by-`n(2)`.

Arguments `kl` and `ku` specify the number of lower and upper off-diagonals, respectively, in `A`. If they are omitted, a full matrix is produced. If only `kl` is present, `ku` defaults to `kl`.

Distribution `mode` can be:

`1`	One large singular value.
`2`	One small singular value.
`3`	Geometrically distributed singular values (default).
`4`	Arithmetically distributed singular values.
`5`	Random singular values with uniformly distributed logarithm.
`< 0`	If `mode` is `-1`, `-2`, `-3`, `-4`, or `-5`, then `randsvd` treats `mode` as `abs(mode)`, except that in the original matrix of singular values the order of the diagonal entries is reversed: small to large instead of large to small.

Condition number `kappa` defaults to `sqrt(1/eps)`. In the special case where `kappa < 0`, `A` is a random, full, symmetric, positive definite matrix with `cond(A) = -kappa` and eigenvalues distributed according to `mode`. Arguments `kl` and `ku`, if present, are ignored.

`A = gallery('randsvd',n,kappa,mode,kl,ku,method)` specifies how the computations are carried out. `method = 0` is the default, while `method = 1` uses an alternative method that is much faster for large dimensions, even though it uses more flops.

redheff — Redheffer's matrix of 1s and 0s

`A = gallery('redheff',n)` returns an `n`-by-`n` matrix of `0`'s and `1`'s defined by `A(i,j) = 1`, if `j = 1` or if `i` divides `j`, and `A(i,j) = 0` otherwise.

The Redheffer matrix has these properties:

- `(n-floor(log2(n)))-1` eigenvalues equal to `1`
- A real eigenvalue (the spectral radius) approximately `sqrt(n)`
- A negative eigenvalue approximately `-sqrt(n)`
- The remaining eigenvalues are provably "small."
- The Riemann hypothesis is true if and only if $\det(A) = O(n^{1/2+\varepsilon})$ for every ε `> 0`.

Barrett and Jarvis conjecture that "the small eigenvalues all lie inside the unit circle `abs(Z) = 1`," and a proof of this conjecture, together with a proof that some eigenvalue tends to zero as `n` tends to infinity, would yield a new proof of the prime number theorem.

riemann — Matrix associated with the Riemann hypothesis

`A = gallery('riemann',n)` returns an `n`-by-`n` matrix for which the Riemann hypothesis is true if and only if

$$\det(A) = O(n!n^{-1/2+\varepsilon})$$

for every $\varepsilon > 0$.

The Riemann matrix is defined by:

```
A = B(2:n+1,2:n+1)
```

where `B(i,j) = i-1` if `i` divides `j`, and `B(i,j) = -1` otherwise.

The Riemann matrix has these properties:

- Each eigenvalue `e(i)` satisfies `abs(e(i)) <= m-1/m`, where `m = n+1`.
- `i <= e(i) <= i+1` with at most `m-sqrt(m)` exceptions.
- All integers in the interval `(m/3, m/2]` are eigenvalues.

ris — Symmetric Hankel matrix

`A = gallery('ris',n)` returns a symmetric `n`-by-`n` Hankel matrix with elements

```
A(i,j) = 0.5/(n-i-j+1.5)
```

The eigenvalues of `A` cluster around $\pi/2$ and $-\pi/2$. This matrix was invented by F.N. Ris.

sampling — Nonsymmetric matrix with ill-conditioned integer eigenvalues.

`A = gallery('sampling',x)`, where `x` is an `n`-vector, is the `n`-by-`n` matrix with `A(i,j) = X(i)/(X(i)-X(j))` for `i ~= j` and `A(j,j)` the sum of the off-diagonal elements in column `j`. A has eigenvalues `0:n-1`. For the eigenvalues 0 and `n`–1, corresponding eigenvectors are `X` and `ones(n,1)`, respectively.

The eigenvalues are ill-conditioned. `A` has the property that `A(i,j) + A(j,i) = 1` for `i ~= j`.

Explicit formulas are available for the left eigenvectors of `A`. For scalar `n`, `sampling(n)` is the same as `sampling(1:n)`. A special case of this matrix arises in sampling theory.

smoke — Complex matrix with a 'smoke ring' pseudospectrum

`A = gallery('smoke',n)` returns an `n`-by-`n` matrix with 1's on the superdiagonal, 1 in the `(n,1)` position, and powers of roots of unity along the diagonal.

`A = gallery('smoke',n,1)` returns the same except that element `A(n,1)` is zero.

The eigenvalues of `gallery('smoke',n,1)` are the `n`th roots of unity; those of `gallery('smoke',n)` are the `n`th roots of unity times `2^(1/n)`.

toeppd — Symmetric positive definite Toeplitz matrix

`A = gallery('toeppd',n,m,w,theta)` returns an `n`-by-`n` symmetric, positive semi-definite (SPD) Toeplitz matrix composed of the sum of `m` rank 2 (or, for certain `theta`, rank 1) SPD Toeplitz matrices. Specifically,

```
T = w(1)*T(theta(1)) + ... + w(m)*T(theta(m))
```

where `T(theta(k))` has `(i,j)` element `cos(2*pi*theta(k)*(i-j))`.

By default: `m = n`, `w = rand(m,1)`, and `theta = rand(m,1)`.

toeppen — Pentadiagonal Toeplitz matrix (sparse)

`P = gallery('toeppen',n,a,b,c,d,e)` returns the `n`-by-`n` sparse, pentadiagonal Toeplitz matrix with the diagonals: `P(3,1) = a`, `P(2,1) = b`, `P(1,1) = c`, `P(1,2) = d`, and `P(1,3) = e`, where `a`, `b`, `c`, `d`, and `e` are scalars.

By default, `(a,b,c,d,e) = (1,-10,0,10,1)`, yielding a matrix of Rutishauser. This matrix has eigenvalues lying approximately on the line segment `2*cos(2*t) + 20*i*sin(t)`.

tridiag — Tridiagonal matrix (sparse)

`A = gallery('tridiag',c,d,e)` returns the tridiagonal matrix with subdiagonal `c`, diagonal `d`, and superdiagonal `e`. Vectors `c` and `e` must have `length(d)-1`.

A = gallery('tridiag',n,c,d,e), where c, d, and e are all scalars, yields the Toeplitz tridiagonal matrix of order n with subdiagonal elements c, diagonal elements d, and superdiagonal elements e. This matrix has eigenvalues

```
d + 2*sqrt(c*e)*cos(k*pi/(n+1))
```

where k = 1:n. (see [1].)

A = gallery('tridiag',n) is the same as A = gallery('tridiag',n,-1,2,-1), which is a symmetric positive definite M-matrix (the negative of the second difference matrix).

triw — Upper triangular matrix discussed by Wilkinson and others

A = gallery('triw',n,alpha,k) returns the upper triangular matrix with ones on the diagonal and alphas on the first k >= 0 superdiagonals.

Order n may be a 2-element vector, in which case the matrix is n(1)-by-n(2) and upper trapezoidal.

Ostrowski ["On the Spectrum of a One-parametric Family of Matrices," *J. Reine Angew. Math.,* 1954] shows that

```
cond(gallery('triw',n,2)) = cot(pi/(4*n))^2,
```

and, for large abs(alpha), cond(gallery('triw',n,alpha)) is approximately abs(alpha)^n*sin(pi/(4*n-2)).

Adding -2^(2-n) to the (n,1) element makes triw(n) singular, as does adding -2^(1-n) to all the elements in the first column.

uniformdata — Array of arbitrary data from standard uniform distribution

A = gallery('uniformdata',[m,n,...],j) returns an m-by-n-by-... array A. The values of A are a random sample from the standard uniform distribution. j must be an integer value in the interval [0, 2^32-1]. Calling gallery('uniformdata', ...) with different values of j will return different arrays. Repeated calls to

gallery('uniformdata',...) with the same size vector and j inputs will always return the same array.

In any call to gallery('uniformdata', ...) you can substitute individual inputs m,n,... for the size vector input [m,n,...]. For example, gallery('uniformdata',[1,2,3,4],5) is equivalent to gallery('uniformdata',1,2,3,4,5).

[A,B,...] = gallery('uniformdata',[m,n,...],j) returns multiple m-by-n-by-... arrays A, B, ..., containing different values.

A = gallery('uniformdata',[m,n,...],j, classname) produces a matrix of class classname. classname must be either 'single' or 'double'.

Generate the arbitrary 6-by-4 matrix of data from the uniform distribution on [0, 1] corresponding to j = 2.

```
x = gallery('uniformdata', [6, 4], 2);
```

Generate the arbitrary 1-by-2-by-3 single array of data from the uniform distribution on [0, 1] corresponding to j = 17.

```
y = gallery('uniformdata', 1, 2, 3, 17, 'single');
```

wathen — Finite element matrix (sparse, random entries)

A = gallery('wathen',nx,ny) returns a sparse, random, n-by-n finite element matrix where n = 3*nx*ny + 2*nx + 2*ny + 1.

Matrix A is precisely the "consistent mass matrix" for a regular nx-by-ny grid of 8-node (serendipity) elements in two dimensions. A is symmetric, positive definite for any (positive) values of the "density," rho(nx,ny), which is chosen randomly in this routine.

A = gallery('wathen',nx,ny,1) returns a diagonally scaled matrix such that

```
0.25 <= eig(inv(D)*A) <= 4.5
```

where D = diag(diag(A)) for any positive integers nx and ny and any densities rho(nx,ny).

wilk — Various matrices devised or discussed by Wilkinson

`gallery('wilk',n)` returns a different matrix or linear system depending on the value of `n`.

`n = 3`	Upper triangular system `Ux=b` illustrating inaccurate solution.
`n = 4`	Lower triangular system `Lx=b`, ill-conditioned.
`n = 5`	`hilb(6)(1:5,2:6)*1.8144`. A symmetric positive definite matrix.
`n = 21`	`W21+`, a tridiagonal matrix. eigenvalue problem. For more detail, see [2].

References

[1] The MATLAB gallery of test matrices is based upon the work of Nicholas J. Higham at the Department of Mathematics, University of Manchester, Manchester, England. Further background can be found in the books *MATLAB Guide, Second Edition*, Desmond J. Higham and Nicholas J. Higham, SIAM, 2005, and *Accuracy and Stability of Numerical Algorithms,* Nicholas J. Higham, SIAM, 1996.

[2] Wilkinson, J. H., *The Algebraic Eigenvalue Problem*, Oxford University Press, London, 1965, p.308.

See Also

`hadamard` | `hilb` | `invhilb` | `magic` | `wilkinson`

Purpose Gamma function

Syntax

```
Y = gamma(X)
```

Definitions The `gamma` function is defined by the integral:

$$\Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dt$$

The `gamma` function interpolates the `factorial` function. For integer `n`:

```
gamma(n+1) = n! = prod(1:n)
```

Description `Y = gamma(X)` returns the `gamma` function at the elements of `X`. `X` must be real.

Algorithms The computation of `gamma` is based on algorithms outlined in [1]. Several different minimax rational approximations are used depending upon the value of `A`.

References

[1] Cody, J., *An Overview of Software Development for Special Functions*, Lecture Notes in Mathematics, 506, Numerical Analysis Dundee, G. A. Watson (ed.), Springer Verlag, Berlin, 1976.

[2] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sec. 6.5.

See Also `gammainc` | `gammaincinv` | `gammaln` | `psi`

gammainc

Purpose

Incomplete gamma function

Syntax

```
Y = gammainc(X,A)
Y = gammainc(X,A,tail)
Y = gammainc(X,A,'scaledlower')
Y = gammainc(X,A,'scaledupper')
```

Definitions

The incomplete `gamma` function is:

$$P(a,x) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$$

For any `A` ≥ 0, `gammainc(X,A)` approaches 1 as `X` approaches `infinity`. For small `X` and `A`, `gammainc(X,A)` is approximately equal to `X^A`, so `gammainc(0,0) = 1`.

Description

`Y = gammainc(X,A)` returns the incomplete `gamma` function of corresponding elements of `X` and `A`. The elements of `A` must be nonnegative. Furthermore, `X` and `A` must be real and the same size (or either can be scalar).

`Y = gammainc(X,A,tail)` specifies the tail of the incomplete `gamma` function. The choices for `tail` are `'lower'` (the default) and `'upper'`. The upper incomplete `gamma` function is defined as:

$$Q(a,x) = \frac{1}{\Gamma(a)} \int_x^{\infty} e^{-t} t^{a-1} dt = 1 - P(a,x).$$

When the upper tail value is close to 0, the `'upper'` option provides a way to compute that value more accurately than by subtracting the lower tail value from 1.

`Y = gammainc(X,A,'scaledlower')` and `Y = gammainc(X,A,'scaledupper')` return the incomplete gamma function, scaled by

$$\Gamma(a+1)\left(\frac{e^x}{x^a}\right).$$

These functions are unbounded above, but are useful for values of `X` and `A` where `gammainc(X,A,'lower')` or `gammainc(X,A,'upper')` underflow to zero.

Note When `X` is negative, `Y` can be inaccurate for `abs(X)>A+1`. This applies to all syntaxes.

References

[1] Cody, J., *An Overview of Software Development for Special Functions*, Lecture Notes in Mathematics, 506, Numerical Analysis Dundee, G. A. Watson (ed.), Springer Verlag, Berlin, 1976.

[2] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sec. 6.5.

See Also

`gamma` | `gammaincinv` | `gammaln` | `psi`

gammaincinv

Purpose

Inverse incomplete gamma function

Syntax

```
x = gammaincinv(y,a)
y = gammaincinv(x,a,tail)
```

Description

`x = gammaincinv(y,a)` evaluates the inverse incomplete gamma function for corresponding elements of `y` and `a`, such that `y = gammainc(x,a)`. The elements of `y` must be in the closed interval `[0,1]`, and those of `a` must be nonnegative. `y` and `a` must be real and the same size (or either can be a scalar).

`y = gammaincinv(x,a,tail)` specifies the tail of the incomplete gamma function. Choices are `lower` (the default) to use the integral from 0 to `x`, or `upper` to use the integral from `x` to infinity.

These two choices are related as:

`gammaincinv(y,a,'upper') = gammaincinv(1-y,a,'lower')`.

When `y` is close to 0, the `upper` option provides a way to compute `x` more accurately than by subtracting `y` from 1.

Definitions

The lower incomplete gamma function is defined as:

$$\mathrm{gammainc}(x,a) = \frac{1}{\Gamma(a)}\int_0^x e^{-t} t^{(a-1)} dt$$

The upper incomplete gamma function is defined as:

$$\mathrm{gammainc}(x,a) = \frac{1}{\Gamma(a)}\int_x^\infty e^{-t} t^{(a-1)} dt$$

`gammaincinv` computes the inverse of the incomplete gamma function with respect to the integration limit `x` using Newton's method.

For any `a>0`, as `y` approaches 1, `gammaincinv(y,a)` approaches infinity.

For small `x` and `a`, `gammainc(x,a)` $\cong x^a$, so `gammaincinv(1,0) = 0`.

References

[1] Cody, J., *An Overview of Software Development for Special Functions*, Lecture Notes in Mathematics, 506, Numerical Analysis Dundee, G. A. Watson (ed.), Springer Verlag, Berlin, 1976.

[2] Abramowitz, M. and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Math. Series #55, Dover Publications, 1965, sec. 6.5.

See Also

`gamma` | `gammainc` | `gammaln` | `psi`

gammaln

Purpose Logarithm of gamma function

Syntax

```
Y = gammaln(A)
```

Description `Y = gammaln(A)` returns the logarithm of the `gamma` function, `gammaln(A) = log(gamma(A))`. The `gammaln` command avoids the underflow and overflow that may occur if it is computed directly using `log(gamma(A))`.

See Also `gammainc` | `gammaincinv` | `gamma` | `psi`

Purpose

Current axes handle

Syntax

```
h = gca
```

Description

`h = gca` returns the handle to the current axes for the current figure. If no axes exists, the MATLAB software creates one and returns its handle. You can use the statement

```
get(gcf,'CurrentAxes')
```

if you do not want MATLAB to create an axes if one does not already exist.

Current Axes

The current axes is the target for graphics output when you create axes children. The current axes is typically the last axes used for plotting or the last axes clicked on by the mouse. Graphics commands such as `plot`, `text`, and `surf` draw their results in the current axes. Changing the current figure also changes the current axes.

See Also

`axes` | `cla` | `gcf` | `findobj` | `CurrentAxes`

gcbf

Purpose

Handle of figure containing object whose callback is executing

Syntax

```
fig = gcbf
```

Description

`fig = gcbf` returns the handle of the figure that contains the object whose callback is currently executing. This object can be the figure itself, in which case, `gcbf` returns the figure's handle.

When no callback is executing, `gcbf` returns the empty matrix, `[]`.

The value returned by `gcbf` is identical to the `figure` output argument returned by `gcbo`.

See Also

`gcbo` | `gco` | `gcf` | `gca`

Purpose

Handle of object whose callback is executing

Syntax

```
h = gcbo
[h,figure] = gcbo
```

Description

`h = gcbo` returns the handle of the graphics object whose callback is executing.

`[h,figure] = gcbo` returns the handle of the current callback object and the handle of the figure containing this object.

Tips

The MATLAB software stores the handle of the object whose callback is executing in the root `CallbackObject` property. If a callback interrupts another callback, MATLAB replaces the `CallbackObject` value with the handle of the object whose callback is interrupting. When that callback completes, MATLAB restores the handle of the object whose callback was interrupted.

The root `CallbackObject` property is read only, so its value is always valid at any time during callback execution. The root `CurrentFigure` property, and the figure `CurrentAxes` and `CurrentObject` properties (returned by `gcf`, `gca`, and `gco`, respectively) are user settable, so they can change during the execution of a callback, especially if that callback is interrupted by another callback. Therefore, those functions are not reliable indicators of which object's callback is executing.

When you write callback routines for the `CreateFcn` and `DeleteFcn` of any object and the figure `ResizeFcn`, you must use `gcbo` since those callbacks do not update the root's `CurrentFigure` property, or the figure's `CurrentObject` or `CurrentAxes` properties; they only update the root's `CurrentFigure` property.

When no callbacks are executing, `gcbo` returns `[]` (an empty matrix).

See Also

`gca` | `gcf` | `gco` | `rootobject`

gcd

Purpose

Greatest common divisor

Syntax

```
G = gcd(A,B)
[G,U,V] = gcd(A,B)
```

Description

`G = gcd(A,B)` returns the greatest common divisors of the elements of `A` and `B`. The elements in `G` are always nonnegative, and `gcd(0,0)` returns `0`. This syntax supports inputs of any numeric type.

`[G,U,V] = gcd(A,B)` also returns the Bézout coefficients, `U` and `V`, which satisfy: `A.*U + B.*V = G`. The Bézout coefficients are useful for solving Diophantine equations. This syntax supports double, single, and signed integer inputs.

Input Arguments

A,B - Input values

scalars, vectors, or arrays of real integer values

Input values, specified as scalars, vectors, or arrays of real integer values. `A` and `B` can be any numeric type, and they can be of different types within certain limitations:

- If `A` or `B` is of type `single`, then the other can be of type `single` or `double`.
- If `A` or `B` belongs to an integer class, then the other must belong to the same class or it must be a `double` scalar value.

`A` and `B` must be the same size or one must be a scalar.

Example: `[20 -3 13],[10 6 7]`

Example: `int16([100 -30 200]),int16([20 15 9])`

Example: `int16([100 -30 200]),20`

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

Output Arguments

G - Greatest common divisor
real, nonnegative integer values

Greatest common divisor, returned as an array of real nonnegative integer values. `G` is the same size as `A` and `B`, and the values in `G` are always real and nonnegative. `G` is returned as the same type as `A` and `B`. If `A` and `B` are of different types, then `G` is returned as the nondouble type.

U,V - Bézout coefficients
real integer values

Bézout coefficients, returned as arrays of real integer values that satisfy the equation, `A.*U + B.*V = G`. The data type of `U` and `V` is the same type as that of `A` and `B`. If `A` and `B` are of different types, then `U` and `V` are returned as the nondouble type.

Algorithms

`g = gcd(A,B)` is calculated using the Euclidian algorithm.[1]

`[g,u,v] = gcd(A,B)` is calculated using the extended Euclidian algorithm.[1]

Examples

Greatest Common Divisors of Double Values

```
A = [-5 17; 10 0];
B = [-15 3; 100 0];
G = gcd(A,B)
```

```
G =

     5     1
    10     0
```

`gcd` returns positive values, even when the inputs are negative.

Greatest Common Divisors of Unsigned Integers

```
A = uint16([255 511 15]);
B = uint16([15 127 1023]);
```

```
G = gcd(A,B)

G =

    15     1     3
```

Solution to Diophantine Equation

Solve the Diophantine equation, $30x + 56y = 8$ for x and y.

Find the greatest common divisor and a pair of Bézout coefficients for `30` and `56`.

```
[g,u,v] = gcd(30,56)

g =

     2

u =

   -13

v =

     7
```

`u` and `v` satisfy the Bézout's identity, `(30*u) + (56*v) = g`.

Rewrite Bézout's identity so that it looks more like the original equation. Do this by multiplying by `4`. Use == to verify that both sides of the equation are equal.

```
(30*u*4) + (56*v*4) == g*4

ans =

   1
```

Calculate the values of *x* and *y* that solve the problem.

```
x = u*4
y = v*4

x =

   -52

y =

    28
```

References

[1] Knuth, D. “Algorithms A and X.” *The Art of Computer Programming, Vol. 2*, Section 4.5.2. Reading, MA: Addison-Wesley, 1973.

See Also

`lcm`

gcf

Purpose

Current figure handle

Syntax

```
h = gcf
```

Description

`h = gcf` returns the handle of the current figure. The current figure is the figure window in which graphics commands such as `plot`, `title`, and `surf` draw their results. If no figure exists, the MATLAB software creates one and returns its handle. You can use the statement

```
get(0,'CurrentFigure')
```

if you do not want MATLAB to create a figure if one does not already exist.

See Also

`clf` | `figure` | `gca` | `CurrentFigure`

Purpose

Handle of current object

Syntax

```
h = gco
h = gco(figure_handle)
```

Description

`h = gco` returns the handle of the current object.

`h = gco(figure_handle)` returns the handle of the current object in the figure specified by `figure_handle`.

Tips

The current object is the last object clicked on, excluding `uimenus`. If the mouse click did not occur over a figure child object, the figure becomes the current object. The MATLAB software stores the handle of the current object in the figure's `CurrentObject` property.

The `CurrentObject` of the `CurrentFigure` does not always indicate the object whose callback is being executed. Interruptions of callbacks by other callbacks can change the `CurrentObject` or even the `CurrentFigure`. Some callbacks, such as `CreateFcn` and `DeleteFcn`, and uimenu `Callback`, intentionally do not update `CurrentFigure` or `CurrentObject`.

`gcbo` provides the only completely reliable way to retrieve the handle to the object whose callback is executing, at any point in the `callback` function, regardless of the type of callback or of any previous interruptions.

See Also

`gca` | `gcbo` | `gcf`

ge

Purpose

Test for greater than or equal to

Syntax

```
A >= B
ge(A, B)
```

Description

`A >= B` compares each element of array `A` with the corresponding element of array `B`, and returns an array with elements set to logical 1 (`true`) where `A` is greater than or equal to `B`, or set to logical 0 (`false`) where `A` is less than `B`. Each input of the expression can be an array or a scalar value.

If both `A` and `B` are scalar (i.e., 1-by-1 matrices), then the MATLAB software returns a scalar value.

If both `A` and `B` are nonscalar arrays, then these arrays must have the same dimensions, and MATLAB returns an array of the same dimensions as `A` and `B`.

If one input is scalar and the other a nonscalar array, then the scalar input is treated as if it were an array having the same dimensions as the nonscalar input array. In other words, if input `A` is the number 100, and `B` is a 3-by-5 matrix, then `A` is treated as if it were a 3-by-5 matrix of elements, each set to 100. MATLAB returns an array of the same dimensions as the nonscalar input array.

`ge(A, B)` is called for the syntax `A >= B` when either `A` or `B` is an object.

Examples

Create two 6-by-6 matrices, `A` and `B`, and locate those elements of `A` that are greater than or equal to the corresponding elements of `B`:

```
A = magic(6);
B = repmat(3*magic(3), 2, 2);

A >= B
ans =
     1     0     0     1     1     1
     0     1     0     1     1     1
     1     0     0     1     1     1
     0     1     1     0     1     0
```

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

See Also

`gt` | `eq` | `le` | `lt` | `ne`

How To

- "Relational Operators"

genpath

Purpose

Generate path string

Syntax

```
p = genpath
p = genpath(folderName)
```

Description

`p = genpath` returns a path string, `p`, that includes all the folders and subfolders below *`matlabroot`*`/toolbox`, including empty subfolders.

`p = genpath(folderName)` returns a path string that includes `folderName` and multiple levels of subfolders below `folderName`. The path string does not include folders named `private`, folders that begin with the `@` character (class folders), folders that begin with the `+` character (package folders), or subfolders within any of these.

Input Arguments

folderName - Folder name

string

Folder name, specified as a string.

Example: `'c:/matlab/myfiles'`

Examples

Add Folder and Subfolders to Search Path

Use `genpath` in conjunction with `addpath` to add a folder and its subfolders to the search path.

Generate a path that includes *`matlabroot`*`/toolbox/images/colorspaces` and all folders below it.

```
folderName = fullfile(matlabroot,'toolbox','images','colorspaces');
p = genpath(folderName);
```

Add the folder and its subfolders to the search path.

```
addpath(p)
```

See Also

`addpath` | `path` | `rmpath`

Concepts

- "What Is the MATLAB Search Path?"

genvarname

Purpose

Construct valid variable name from string

Syntax

```
varname = genvarname(str)
varname = genvarname(str, exclusions)
```

Description

`varname = genvarname(str)` constructs a string `varname` that is similar to or the same as the `str` input, and can be used as a valid variable name. `str` can be a single character array or a cell array of strings. If `str` is a cell array of strings, `genvarname` returns a cell array of strings in `varname`. The strings in a cell array returned by `genvarname` are guaranteed to be different from each other.

`varname = genvarname(str, exclusions)` returns a valid variable name that is different from any name listed in the `exclusions` input. The `exclusions` input can be a single character array or a cell array of strings. Specify the function `who` in the `exclusions` character array to create a variable name that will be unique in the current MATLAB workspace (see "Example 4" on page 1-1948, below).

Note `genvarname` returns a string that can be used as a variable name. It does not create a variable in the MATLAB workspace. You cannot, therefore, assign a value to the output of `genvarname`.

Tips

A valid MATLAB variable name is a character string of letters, digits, and underscores, such that the first character is a letter, and the length of the string is less than or equal to the value returned by the `namelengthmax` function. Any string that exceeds `namelengthmax` is truncated in the `varname` output. See "Example 6" on page 1-1949, below.

The variable name returned by `genvarname` is not guaranteed to be different from other variable names currently in the MATLAB workspace unless you use the `exclusions` input in the manner shown in "Example 4" on page 1-1948, below.

If you use `genvarname` to generate a field name for a structure, MATLAB does create a variable for the structure and field in the MATLAB workspace. See "Example 3" on page 1-1947, below.

If the `str` input contains any whitespace characters, `genvarname` removes then and capitalizes the next alphabetic character in `str`. If `str` contains any nonalphanumeric characters, `genvarname` translates these characters into their hexadecimal value.

Examples

Example 1

Create four similar variable name strings that do not conflict with each other:

```
v = genvarname({'A', 'A', 'A', 'A'})
v =
    'A'    'A1'    'A2'    'A3'
```

Example 2

Read a column header `hdr` from worksheet `trial2` in Excel spreadsheet myproj_apr23:

```
[data hdr] = xlsread('myproj_apr23.xls', 'trial2');
```

Make a variable name from the text of the column header that will not conflict with other names:

```
v = genvarname(['Column ' hdr{1,3}]);
```

Assign data taken from the spreadsheet to the variable in the MATLAB workspace:

```
eval([v '= data(1:7, 3);']);
```

Example 3

Collect readings from an instrument once every minute over the period of an hour into different fields of a structure. `genvarname` not only generates unique fieldname strings, but also creates the structure and fields in the MATLAB workspace:

```
for k = 1:60
record.(genvarname(['reading' datestr(clock, 'HHMMSS')])) = takeReading;
pause(60)
end
```

After the program ends, display the recorded data from the workspace:

```
record
record =
    reading090446: 27.3960
    reading090546: 23.4890
    reading090646: 21.1140
    reading090746: 23.0730
    reading090846: 28.5650
          .
          .
          .
```

Example 4

Generate variable names that are unique in the MATLAB workspace by putting the output from the `who` function in the `exclusions` list.

```
for k = 1:5
   t = clock;
   pause(uint8(rand * 10));
   v = genvarname('time_elapsed', who);
   eval([v ' = etime(clock,t)'])
   end
```

As this code runs, you can see that the variables created by `genvarname` are unique in the workspace:

```
time_elapsed =
    5.0070
time_elapsed1 =
    2.0030
time_elapsed2 =
    7.0010
```

```
time_elapsed3 =
    8.0010
time_elapsed4 =
    3.0040
```

After the program completes, use the `who` function to view the workspace variables:

```
who

k            time_elapsed   time_elapsed2  time_elapsed4
t            time_elapsed1  time_elapsed3  v
```

Example 5

If you try to make a variable name from a MATLAB keyword, `genvarname` creates a variable name string that capitalizes the keyword and precedes it with the letter `x`:

```
v = genvarname('global')
v =
   xGlobal
```

Example 6

If you enter a string that is longer than the value returned by the `namelengthmax` function, `genvarname` truncates the resulting variable name string:

```
namelengthmax
ans =
    63

vstr = genvarname(sprintf('%s%s', ...
   'This name truncates because it contains ', ...
   'more than the maximum number of characters'))
vstr =
ThisNameTruncatesBecauseItContainsMoreThanTheMaximumNumberOfCha
```

See Also

`isvarname` | `iskeyword` | `isletter` | `namelengthmax` | `who` | `regexp`

Purpose Query Handle Graphics object properties

Syntax

```
get(h)
get(h,'PropertyName')
<m-by-n value cell array> = get(H,pn)
a = get(h)
a = get(0)
a = get(0,'Factory')
a = get(0,'FactoryObjectTypePropertyName')
a = get(h,'Default')
a = get(h,'DefaultObjectTypePropertyName')
```

Description

Note Do not use the `get` function on Java objects as it will cause a memory leak. For more information, see "Accessing Private and Public Data"

`get(h)` returns all properties of the graphics object identified by the handle `h` and their current values. For this syntax, `h` must be a scalar.

`get(h,'PropertyName')` returns the value of the property `'PropertyName'` of the graphics object identified by `h`.

`<m-by-n value cell array> = get(H,pn)` returns *n* property values for *m* graphics objects in the *m*-by-*n* cell array, where `m = length(H)` and *n* is equal to the number of property names contained in `pn`.

`a = get(h)` returns a structure whose field names are the object's property names and whose values are the current values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen. For this syntax, `h` may be a scalar or a `m`-by-`n` array of handles. If `h` is a vector, `a` will be a `(m*n)`-by-1 struct array.

`a = get(0)` returns the current values of all user-settable properties. `a` is a structure array whose field names are the object property names and whose field values are the values of the corresponding properties.

If you do not specify an output argument, MATLAB displays the information on the screen.

`a = get(0,'Factory')` returns the factory-defined values of all user-settable properties. `a` is a structure array whose field names are the object property names and whose field values are the values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen.

`a = get(0,'FactoryObjectTypePropertyName')` returns the factory-defined value of the named property for the specified object type. The argument *FactoryObjectTypePropertyName* is the word `Factory` concatenated with the object type (e.g., `Figure`) and the property name (e.g., `Color`)`FactoryFigureColor`.

`a = get(h,'Default')` returns all default values currently defined on object `h`. `a` is a structure array whose field names are the object property names and whose field values are the values of the corresponding properties. If you do not specify an output argument, MATLAB displays the information on the screen.

`a = get(h,'DefaultObjectTypePropertyName')` returns the factory-defined value of the named property for the specified object type. The argument *DefaultObjectTypePropertyName* is the word `Default` concatenated with the object type (e.g., `Figure`) and the property name (e.g., `Color`).

```
DefaultFigureColor
```

Examples

You can obtain the default value of the `LineWidth` property for line graphics objects defined on the root level with the statement

```
get(0,'DefaultLineLineWidth')
ans =
      0.5000
```

To query a set of properties on all axes children, define a cell array of property names:

```
props = {'HandleVisibility', 'Interruptible';
```

```
   'SelectionHighlight', 'Type'};
output = get(get(gca,'Children'),props);
```

The variable `output` is a cell array of dimension `length(get(gca,'Children')`-by-4.

For example, type

```
patch;surface;text;line
output = get(get(gca,'Children'),props)
output =
 'on'    'on'    'on'    'line'
 'on'    'off'   'on'    'text'
 'on'    'on'    'on'    'surface'
 'on'    'on'    'on'    'patch'
```

See Also

`findobj` | `gca` | `gcf` | `gco` | `set`

Purpose

Query property values for `audioplayer` object

Syntax

```
Value = get(obj,Name)
Values = get(obj,{Name1,...,NameN})
Values = get(obj)
get(obj)
```

Description

`Value = get(obj,Name)` returns the value of the specified property for object `obj`.

`Values = get(obj,{Name1,...,NameN})` returns the values of the specified properties in a 1-by-`N` cell array.

`Values = get(obj)` returns a scalar structure that contains the values of all properties of `obj`. Each field name corresponds to a property name.

`get(obj)` displays all property names and their current values.

Examples

Create an `audioplayer` object from the example file `handel.mat` and query the object properties:

```
load handel.mat;
handelObj = audioplayer(y, Fs);

% Display all properties.
get(handelObj)

% Display only the SampleRate property.
get(handelObj, 'SampleRate')

% Create a cell array that contains
% values for two properties.
info = get(handelObj, {'BitsPerSample', 'NumberOfChannels'});
```

Alternatives

To access a single property, you can use dot notation. Reference each property as though it is a field of a structure array. For example, find the value of the `TotalSamples` property for an object named `handelObj` (as created in the Example):

```
numSamples = handelObj.TotalSamples;
```

This command is exactly equivalent to:

```
numSamples = get(handelObj, 'TotalSamples');
```

See Also

audioplayer | set

Purpose

Query property values for `audiorecorder` object

Syntax

```
Value = get(obj,Name)
Values = get(obj,{Name1,...,NameN})
Values = get(obj)
get(obj)
```

Description

`Value = get(obj,Name)` returns the value of the specified property for object *obj*.

`Values = get(obj,{Name1,...,NameN})` returns the values of the specified properties in a 1-by-*N* cell array.

`Values = get(obj)` returns a scalar structure that contains the values of all properties of *obj*. Each field name corresponds to a property name.

`get(obj)` displays all property names and their current values.

Examples

Create an `audiorecorder` object and query the object properties:

```
recorderObj = audiorecorder;

% Display all properties.
get(recorderObj)

% Display only the SampleRate property.
get(recorderObj, 'SampleRate')

% Create a cell array that contains
% values for two properties.
info = get(recorderObj, {'BitsPerSample', 'NumberOfChannels'});
```

Alternatives

To access a single property, you can use dot notation. Reference each property as though it is a field of a structure array. For example, find the value of the `TotalSamples` property for an object named `recorderObj` (as created in the Example):

```
numSamples = recorderObj.TotalSamples;
```

This command is exactly equivalent to:

```
numSamples = get(recorderObj, 'TotalSamples');
```

See Also

`audiorecorder` | `set`

Purpose

Get property value from interface, or display properties

Syntax

```
V = h.get
V = h.get('propertyname')
V = get(h,...)
```

Description

`V = h.get` returns a list of all properties and their values for the object or interface, `h`.

If `V` is empty, either there are no properties in the object, or the MATLAB software cannot read the object's type library. Refer to the COM vendor's documentation. For Automation objects, if the vendor provides documentation for a specific property, use the `V = get(h,...)` syntax to call it.

`V = h.get('propertyname')` returns the value of the property specified in the string, `propertyname`.

`V = get(h,...)` is an alternate syntax for the same operation.

Tips

The meaning and type of the return value is dependent upon the specific property being retrieved. The object's documentation should describe the specific meaning of the return value. MATLAB may convert the data type of the return value. For a description of how MATLAB converts COM data types, see "Handling COM Data in MATLAB Software".

COM functions are available on Microsoft Windows systems only.

Examples

Create a COM server running Microsoft Excel software:

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

Retrieve a single property value:

```
e.Path
```

Depending on your spreadsheet program, MATLAB software displays:

```
ans =
```

```
C:\Program Files\MSOffice\OFFICE11
```

Retrieve a list of all properties for the `CommandBars` interface:

```
c = e.CommandBars.get
```

MATLAB displays information similar to the following:

```
c =
                   Application: [1x1
Interface.Microsoft_Excel_11.0_Object_Library._Application]
                       Creator: 1.4808e+009
                 ActionControl: []
                 ActiveMenuBar: [1x1
Interface.Microsoft_Office_12.0_Object_Library.CommandBar]
                         Count: 129
               DisplayTooltips: 1
         DisplayKeysInTooltips: 0
                  LargeButtons: 0
            MenuAnimationStyle: 'msoMenuAnimationNone'
                        Parent: [1x1
Interface.Microsoft_Excel_11.0_Object_Library._Application]
                 AdaptiveMenus: 0
                  DisplayFonts: 1
              DisableCustomize: 0
    DisableAskAQuestionDropdown: 0
```

See Also

`set (COM)` | `inspect` | `isprop` | `addproperty` | `deleteproperty`

Purpose

Query property values of handle objects derived from `hgsetget` class

Syntax

```
CV = get(H,'PropertyName')
SV = get(h)
get(h)
```

Description

`CV = get(H,'PropertyName')` returns the value of the named property from the objects in the handle array `H`. If `H` is scalar, `get` returns a single value; if `H` is an array, `get` returns a cell array of property values. If you specify a cell array of property names, then `get` returns a cell array of values, where each row in the cell corresponds to an element in `H` and each column in the cell corresponds to an element in the property name cell array.

If `H` is nonscalar and `PropertyName` is the name of a dynamic property, `get` returns a value only if the property exists in all objects referenced in `H`.

`SV = get(h)` returns a `struct` array in which the field names are the object's property names and the values are the current values of the corresponding properties. If `h` is an array, then `SV` is a `numel(h)`-by-`1` array of `struct`s.

`get(h)` displays all property names and their current values for the MATLAB objects with handle `h`.

Your subclass can override the `hgsetget` `getdisp` method to control how MATLAB displays this information.

See Also

`get` | `handle` | `hgsetget` | `set (hgsetget)`

How To

- "Implementing a Set/Get Interface for Properties"

get (memmapfile)

Purpose

Memmapfile object properties

Syntax

```
s = get(obj)
val = get(obj, prop)
```

Description

`s = get(obj)` returns the values of all properties of the `memmapfile` object `obj` in structure array `s`. Each property retrieved from the object is represented by a field in the output structure. The name and contents of each field are the same as the name and value of the property it represents.

Note Although property names of a `memmapfile` object are not case sensitive, field names of the output structure returned by `get` (named the same as the properties they represent) are case sensitive.

`val = get(obj, prop)` returns the value(s) of one or more properties specified by `prop`. The `prop` input can be a quoted string or a cell array of quoted strings, each containing a property name. If the latter is true, `get` returns the property values in a cell array.

Examples

You can use the `get` method of the `memmapfile` class to return information on any or all of the object's properties. Specify one or more property names to get the values of specific properties.

This example returns the values of the `Offset`, `Repeat`, and `Format` properties for a `memmapfile` object. Start by constructing the object:

```
m = memmapfile('records.dat',                    ...
               'Offset', 2048,                   ...
               'Format', {                       ...
                  'int16'  [2 2] 'model';        ...
                  'uint32' [1 1] 'serialno';     ...
                  'single' [1 3] 'expenses'});
```

Use the get method to return the specified property values in a 1-by-3 cell array m_props:

```
m_props = get(m, {'Offset', 'Repeat', 'Format'})
m_props =
    [2048]    [Inf]    {3x3 cell}

m_props{3}
ans =
    'int16'     [1x2 double]    'model'
    'uint32'    [1x2 double]    'serialno'
    'single'    [1x2 double]    'expenses'
```

Another way to return the same information is to use the objname.property syntax:

```
m_props = {m.Offset, m.Repeat, m.Format}
m_props =
    [2048]    [Inf]    {3x3 cell}
```

To return the values for all properties with get, pass just the object name:

```
s = get(m)
    Filename: 'd:\matlab\records.dat'
    Writable: 0
      Offset: 2048
      Format: {3x3 cell}
      Repeat: Inf
        Data: [753 1]
```

To see just the Format field of the returned structure, type

```
s.Format
ans =
    'int16'     [1x2 double]    'model'
    'uint32'    [1x2 double]    'serialno'
    'single'    [1x2 double]    'expenses'
```

See Also

memmapfile | disp (memmapfile)

Purpose

Query property values for video reader object

Syntax

```
Value = get(obj,Name)
Values = get(obj,{Name1,...,NameN})
allValues = get(obj)
get(obj)
```

Description

`Value = get(obj,Name)` returns the value of the property with the specified `Name` for object `obj`.

`Values = get(obj,{Name1,...,NameN})` returns the values of the specified properties in a 1-by-`N` cell array.

`allValues = get(obj)` returns a scalar structure that contains the values of all properties of `obj`. Each field name corresponds to a property name.

`get(obj)` displays all property names and their current values.

Input Arguments

obj

`VideoReader` object created by the `VideoReader` function.

Name

String enclosed in single quotation marks that specifies a `VideoReader` property.

Output Arguments

Value

String containing the value associated with the specified `VideoReader` property.

Values

Cell array containing the values associated with the specified `VideoReader` properties. Values is a row vector, with one column for each property.

allValues

Scalar (1-by-1) structure array that contains the values of all properties of `VideoReader` object `obj`.

Examples

Display all properties of an object associated with the example file `xylophone.mpg`:

```
xyloObj = VideoReader('xylophone.mpg');
get(xyloObj)
```

Create a 1-by-3 cell array that contains the values of the `Height`, `Width`, and `NumberOfFrames` properties of an object associated with `xylophone.mpg`:

```
xyloObj = VideoReader('xylophone.mpg');
xyloSize = get(xyloObj, {'Height', 'Width', 'NumberOfFrames'})
```

Alternatives

To access a single property, you can use dot notation. Reference each property as though it is a field of a structure array. For example, find the value of the `NumberOfFrames` property for object `xyloObj` (as created in the Examples):

```
numFrames = xyloObj.NumberOfFrames;
```

This command is exactly equivalent to:

```
numFrames = get(xyloObj, 'NumberOfFrames');
```

See Also

`VideoReader` | `set`

Purpose Random stream properties

Class `@RandStream`

Syntax

```
get(s)
P = get(s)
P = get(s,'PropertyName')
```

Description

`get(s)` prints the list of properties for the random stream `s`.

`P = get(s)` returns all properties of `s` in a scalar structure.

`P = get(s,'PropertyName')` returns the property `'PropertyName'`.

See Also `RandStream` | `set (RandStream)`

get (serial)

Purpose

Serial port object properties

Syntax

```
get(obj)
out = get(obj)
out = get(obj,'PropertyName')
```

Description

`get(obj)` returns all property names and their current values to the command line for the serial port object, `obj`.

`out = get(obj)` returns the structure `out` where each field name is the name of a property of `obj`, and each field contains the value of that property.

`out = get(obj,'PropertyName')` returns the value `out` of the property specified by *`PropertyName`* for `obj`. If *`PropertyName`* is replaced by a 1-by-n or n-by-1 cell array of strings containing property names, then `get` returns a 1-by-n cell array of values to `out`. If `obj` is an array of serial port objects, then `out` will be a m-by-n cell array of property values where m is equal to the length of `obj` and n is equal to the number of properties specified.

Tips

Refer to Displaying Property Names and Property Values for a list of serial port object properties that you can return with `get`.

When you specify a property name, you can do so without regard to case, and you can make use of property name completion. For example, if `s` is a serial port object, then these commands are all valid.

```
out = get(s,'BaudRate');
out = get(s,'baudrate');
out = get(s,'BAUD');
```

If you use the `help` command to display help for `get`, then you need to supply the pathname shown below.

```
help serial/get
```

Examples

This example illustrates some of the ways you can use `get` to return property values for the serial port object `s` on a Windows platform.

```
s = serial('COM1');
out1 = get(s);
out2 = get(s,{'BaudRate','DataBits'});
get(s,'Parity')
ans =
none
```

See Also

`set`

get (timer)

Purpose

Timer object properties

Syntax

```
get(obj)
V = get(obj)
V = get(obj,'PropertyName')
```

Description

`get(obj)` displays all property names and their current values for the timer object `obj`. `obj` must be a single timer object.

`V = get(obj)` returns a structure, `V`, where each field name is the name of a property of `obj` and each field contains the value of that property. If `obj` is an M-by-1 vector of timer objects, V is an M-by-1 array of structures.

`V = get(obj,'PropertyName')` returns the value, `V`, of the timer object property specified in *`PropertyName`*.

If *`PropertyName`* is a 1-by-N or N-by-1 cell array of strings containing property names, `V` is a 1-by-N cell array of values. If `obj` is a vector of timer objects, `V` is an M-by-N cell array of property values where M is equal to the length of `obj` and N is equal to the number of properties specified.

Examples

```
t = timer;
get(t)
       AveragePeriod: NaN
            BusyMode: 'drop'
            ErrorFcn: ''
       ExecutionMode: 'singleShot'
       InstantPeriod: NaN
                Name: 'timer-1'
    ObjectVisibility: 'on'
              Period: 1
             Running: 'off'
          StartDelay: 1
            StartFcn: ''
             StopFcn: ''
                 Tag: ''
```

```
        TasksExecuted: 0
       TasksToExecute: Inf
             TimerFcn: ''
                 Type: 'timer'
             UserData: []
get(t, {'StartDelay','Period'})
ans =

    [0]    [1]
```

See Also

`timer` | `set(timer)`

get (tscollection)

Purpose

Query `tscollection` object property values

Syntax

```
value = get(tsc,'PropertyName')
```

Description

`value = get(tsc,'PropertyName')` returns the value of the specified property of the `tscollection` object tsc. The following syntax is equivalent:

```
value = tsc.PropertyName
```

`get(tsc)` displays all properties and values of the `tscollection` object tsc.

See Also

`set (tscollection)` | `timeseries`

Purpose

Extract date-string time vector into cell array

Syntax

```
getabstime(tsc)
```

Description

`getabstime(tsc)` extracts the time vector from the `tscollection` object tsc as a cell array of date strings. To define the time vector relative to a calendar date, set the `TimeInfo.StartDate` property of the time-series collection. When the `TimeInfo.StartDate` format is a valid `datestr` format, the output strings from `getabstime` have the same format.

Examples

1 Create a `tscollection` object.

```
tsc = tscollection(timeseries([3 6 8 0 10]));
```

2 Set the `StartDate` property.

```
tsc.TimeInfo.StartDate = '10/27/2005 07:05:36';
```

3 Extract a vector of absolute time values.

```
getabstime(tsc)

ans =

    '27-Oct-2005 07:05:36'
    '27-Oct-2005 07:05:37'
    '27-Oct-2005 07:05:38'
    '27-Oct-2005 07:05:39'
    '27-Oct-2005 07:05:40'
```

4 Change the date-string format of the time vector.

```
tsc.TimeInfo.Format = 'mm/dd/yy';
```

5 Extract the time vector with the new date-string format.

```
getabstime(tsc)
```

```
ans =

    '10/27/05'
    '10/27/05'
    '10/27/05'
    '10/27/05'
    '10/27/05'
```

See Also

datestr | tscollection | setabstime (tscollection)

Purpose

Value of application-defined data

Syntax

```
value = getappdata(h,'name')
values = getappdata(h)
```

Description

`value = getappdata(h,'name')` gets the value of the application-defined data with the name specified by `name`, in the object with handle `h`. If the application-defined data does not exist, the MATLAB software returns an empty matrix in `value`.

`values = getappdata(h)` returns all application-defined data for the object with handle `h`.

Tips

Application data is data that is meaningful to or defined by your application which you attach to a figure or any GUI component (other than ActiveX controls) through its `AppData` property. Only Handle Graphics MATLAB objects use this property.

See Also

`setappdata` | `rmappdata` | `isappdata`

audiorecorder.getaudiodata

Purpose

Store recorded audio signal in numeric array

Syntax

```
y = getaudiodata(recorder)
y = getaudiodata(recorder, dataType)
```

Description

`y = getaudiodata(recorder)` returns recorded audio data associated with `audiorecorder` object *recorder* to `double` array *y*.

`y = getaudiodata(recorder, dataType)` converts the signal data to the specified data type: `'double'`, `'single'`, `'int16'`, `'int8'`, or `'uint8'`.

Output Arguments

y

Audio signal data *y* contains the same number of columns as the number of channels in the recording: one for mono, two for stereo. The range of values depends on the data type, as shown in the following table.

Data Type	Sample Value Range
`int8`	-128 to 127
`uint8`	0 to 255
`int16`	-32768 to 32767
`single`	-1 to 1
`double`	-1 to 1

Examples

Collect a sample of your speech with a microphone, and plot the signal data:

```
% Record your voice for 5 seconds.
recObj = audiorecorder;
disp('Start speaking.')
recordblocking(recObj, 5);
disp('End of Recording.');
```

```
% Play back the recording.
play(recObj);

% Store data in double-precision array.
myRecording = getaudiodata(recObj);

% Plot the waveform.
plot(myRecording);
```

See Also

`audiorecorder`

How To

- “Characteristics of Audio Files”
- “Record Audio”

GetCharArray

Purpose

Character array from Automation server

Syntax

MATLAB Client

```
str = h.GetCharArray('varname', 'workspace')
str = GetCharArray(h, 'varname', 'workspace')
```

IDL Method Signature

```
HRESULT GetCharArray([in] BSTR varName, [in] BSTR Workspace,  [out, retva
```

Microsoft Visual Basic Client

```
GetCharArray(varname As String, workspace As String) As String
```

Description

`str = h.GetCharArray('varname', 'workspace')` gets the character array stored in `varname` from the specified `workspace` of the server attached to handle `h` and returns it in `str`. The values for *workspace* arebase or `global`.

`str = GetCharArray(h, 'varname', 'workspace')` is an alternate syntax.

Examples

This example uses a MATLAB client.

```
h = actxserver('matlab.application');
%Assign a string to variable 'str' in the base workspace of the server
h.PutCharArray('str', 'base', ...
    'He jests at scars that never felt a wound.');
%Read 'str' back in the client
S = h.GetCharArray('str', 'base')
```

This example uses a Visual Basic .NET client.

1. Create the Visual Basic application. Use the `MsgBox` command to control flow between MATLAB and the application.

   ```
   Dim Matlab As Object
   Dim S As String
   ```

```
Matlab = CreateObject("matlab.application")
MsgBox("In MATLAB, type" & vbCrLf _
    & "str='new string';")
```

2. Open the MATLAB window, then type:

```
str='new string';
```

3. Click **Ok**.

```
Try
    S = Matlab.GetCharArray("str", "base")
    MsgBox("str = " & S)
Catch ex As Exception
    MsgBox("You did not set 'str' in MATLAB")
End Try
```

The Visual Basic `MsgBox` displays what you typed in MATLAB.

See Also

`PutCharArray` | `GetWorkspaceData` | `GetVariable`

RandStream.getDefaultStream

Purpose

Random number stream

Note `RandStream.getDefaultStream` will be removed in a future version. Use `RandStream.getGlobalStream` instead. The shared global stream used by `rand`, `randi`, and `randn`, referred to in former versions of MATLAB as the *default stream* is now referred to as the *global stream*.

Class

`RandStream`

Syntax

```
stream = RandStream.getDefaultStream
```

Description

`stream = RandStream.getDefaultStream` returns the global random number stream. The MATLAB functions `rand`, `randi`, and `randn` use the global stream to generate values.

Purpose Override to change command window display

Syntax

```
getdisp(H)
```

Description `getdisp(H)` called by `get` when `get` is called with no output arguments and a single input argument that is a handle array. Override this `hgsetget` class method in a subclass to change how property information is displayed in the command window.

See Also `hgsetget` | `get (hgsetget)`

How To

- “Implementing a Set/Get Interface for Properties”

getenv

Purpose

Environment variable

Syntax

```
getenv 'name'
N = getenv('name')
```

Description

`getenv 'name'` searches the underlying operating system's environment list for a string of the form `name=value`, where `name` is the input string. If found, the MATLAB software returns the string `value`. If the specified name cannot be found, an empty matrix is returned.

`N = getenv('name')` returns `value` to the variable `N`.

Examples

```
os = getenv('OS')

os =
Windows_NT
```

See Also

`setenv` | `computer` | `pwd` | `ver` | `path`

Purpose

Field of structure array

Syntax

```
value = getfield(struct, 'field')
value =
getfield(struct,{sIndx1,...,sIndxM},'field',{fIndx1,
   ...,fIndxN})
```

Description

`value = getfield(struct, 'field')`, where `struct` is a 1-by-1 structure, returns the contents of the specified field, equivalent to `value = struct.field`. Pass field references as strings.

`value = getfield(struct,{sIndx1,...,sIndxM},'field',{fIndx1,...,fIndxN})` returns the contents of the specified field, equivalent to `value = struct(sIndx1,...,sIndxM).field(fIndx1,...,fIndxN)`. The `getfield` function supports multiple sets of `field` and `fIndx` inputs, and all `Indx` inputs are optional. If structure `struct` or any of the fields is a nonscalar structure, and you do not specify an `Indx`, the `getfield` function returns the values associated with the first index. If you specify a single colon operator for an `fIndx` input, enclose it in single quotation marks: `':'`.

Tips

- For most cases, retrieve data from a structure array by indexing rather than using the `getfield` function. For more information, see "Access Data in a Structure Array" and "Generate Field Names from Variables".
- Call `getfield` to simplify references to structure arrays with nested fields, or to avoid creating unnecessary temporary variables, as shown in the Examples section.

Examples

The `what` function returns a structure array that describes the MATLAB files in the current folder. Find the files with the `.m` extension:

```
files = getfield(what, 'm');
```

To perform the same task by indexing requires that you create a temporary variable:

```
templist = what;
files = templist.m;
```

Find values within a structure that contains nested fields:

```
level = 5;
semester = 'Fall';
subject = 'Math';
student = 'John_Doe';
fieldnames = {semester subject student};

% Add data to a structure named grades.
grades(level).(semester).(subject).(student)(10,21:30) = ...
             [85, 89, 76, 93, 85, 91, 68, 84, 95, 73];

% Retrieve the data added.
getfield(grades, {level}, fieldnames{:}, {10,21:30})
```

Using the structure defined in the previous example, find all values in the tenth row of the specified field:

```
getfield(grades, {level}, fieldnames{:}, {10,':'})
```

See Also

`setfield` | `fieldnames` | `isfield` | `orderfields` | `rmfield`

How To

- "Generate Field Names from Variables"
- "Access Data in a Structure Array"

Purpose

File formats that `VideoReader` supports

Syntax

```
formats = VideoReader.getFileFormats()
```

Description

`formats = VideoReader.getFileFormats()` returns an array of `audiovideo.FileFormatInfo` objects that indicate which formats `VideoReader` can read on the current system. Each object has the following properties: `Extension`, `Description`, `ContainsVideo`, and `ContainsAudio`.

Tips

- On Windows and UNIX systems, the list of file formats does not always contain all the formats that `VideoReader` can read on your system. `getFileFormats` returns a platform-dependent, static list of formats that `VideoReader` can read on most systems.
- On all systems, `VideoReader` cannot always read a particular video file even if `getFileFormats` lists its format. For more information, see Supported Video File Formats.

Output Arguments

formats

Array of `audiovideo.FileFormatInfo` objects, which have the following properties:

`Extension`	File extension.
`Description`	Text description of the file format.
`ContainsVideo`	Whether `VideoReader` can read video from this format.
`ContainsAudio`	Whether `VideoReader` can read audio from this format.

To convert this array to a filter list for dialog boxes generated with `uigetfile`, use the `getFilterSpec` method with the following syntax:

```
filterSpec = getFilterSpec(formats)
```

The filter list includes `'All Video Files'` in the first row of the cell array, and `'All Files (*.*)'` in the last row.

Examples

View the list of file formats that `VideoReader` supports on your system:

```
VideoReader.getFileFormats()
```

On a Windows system, this list appears as follows:

```
Video File Formats:
    .asf - ASF File
    .asx - ASX File
    .avi - AVI File
    .mj2 - Motion JPEG2000
    .mpg - MPEG-1
    .wmv - Windows Media Video
```

Create a dialog box to select a video file:

```
% Get the supported file formats.
formats = VideoReader.getFileFormats();

% Convert to a filter list.
filterSpec = getFilterSpec(formats);

% Create the dialog box.
[filename, pathname] = uigetfile(filterSpec);
```

Check whether `VideoReader` can read an `.avi` file on the current system:

```
fmtList = VideoReader.getFileFormats();
```

```
if any(ismember({fmtList.Extension},'avi'))
     disp('VideoReader can read AVI files on this system.');
else
     disp('VideoReader cannot read AVI files on this system.');
end
```

See Also

VideoReader

getframe

Purpose

Capture movie frame

Syntax

```
getframe
F = getframe
F = getframe(h)
F = getframe(h,rect)
```

Description

`getframe` returns a movie frame. The frame is a snapshot (pixmap) of the current axes or figure.

`F = getframe` gets a frame from the current axes.

`F = getframe(h)` gets a frame from the figure or axes identified by handle `h`.

`F = getframe(h,rect)` specifies a rectangular area from which to copy the pixmap. `rect` is relative to the lower left corner of the figure or axes `h`, in pixel units. `rect` is a four-element vector in the form `[left bottom width height]`, where `width` and `height` define the dimensions of the rectangle.

`getframe` returns a movie frame, which is a structure having two fields:

- `cdata` — The image data stored as a matrix of `uint8` values. The dimensions of `F.cdata` are height-by-width-by-3.
- `colormap` — The colormap stored as an n-by-3 matrix of doubles. `F.colormap` is empty on true color systems.

To capture an image, use this approach:

```
F = getframe(gcf);
image(F.cdata)
colormap(F.colormap)
```

Tips

`getframe` is usually used in a `for` loop to assemble an array of movie frames for playback using `movie`. For example,

```
for j = 1:n
   plotting commands
   F(j) = getframe;
```

```
end
movie(F)
```

If you are capturing frames of a plot that takes a long time to generate or are repeatedly calling `getframe` in a loop, make sure that your computer's screen saver does not activate and that your monitor does not turn off for the duration of the capture; otherwise one or more of the captured frames can contain graphics from your screen saver or nothing at all.

Note In situations where MATLAB software is running on a virtual desktop that is not currently visible on your monitor, calls to `getframe` will complete, but will capture a region on your monitor that corresponds to the position occupied by the figure or axes on the hidden desktop. Therefore, make sure that the window to be captured by `getframe` exists on the currently active desktop.

Capture Regions

`F = getframe` returns the contents of the current axes, exclusive of the axis labels, title, or tick labels.

`F = getframe(gcf)` captures the entire interior of the current figure window.

To capture the figure window menu, use `F = getframe(h,rect)` with a rectangle sized to include the menu.

Resolution of Captured Frames

The resolution of the framed image depends on the size of the axes in pixels when `getframe` is called. As the `getframe` command takes a snapshot of the screen, if the axes is small in size (e.g., because you have restricted the view to a window within the axes), `getframe` captures fewer screen pixels, and the captured image might have poor resolution if enlarged for display.

Capturing UIControls

To capture uicontrols along with the axes and any annotations displayed on the plot, specify the figure handle:

```
F = getframe(figure_handle);
```

To exclude uicontrols outside of the current axes, do not specify the figure handle:

```
F = getframe;
```

Limitations with Renderer on Windows Systems

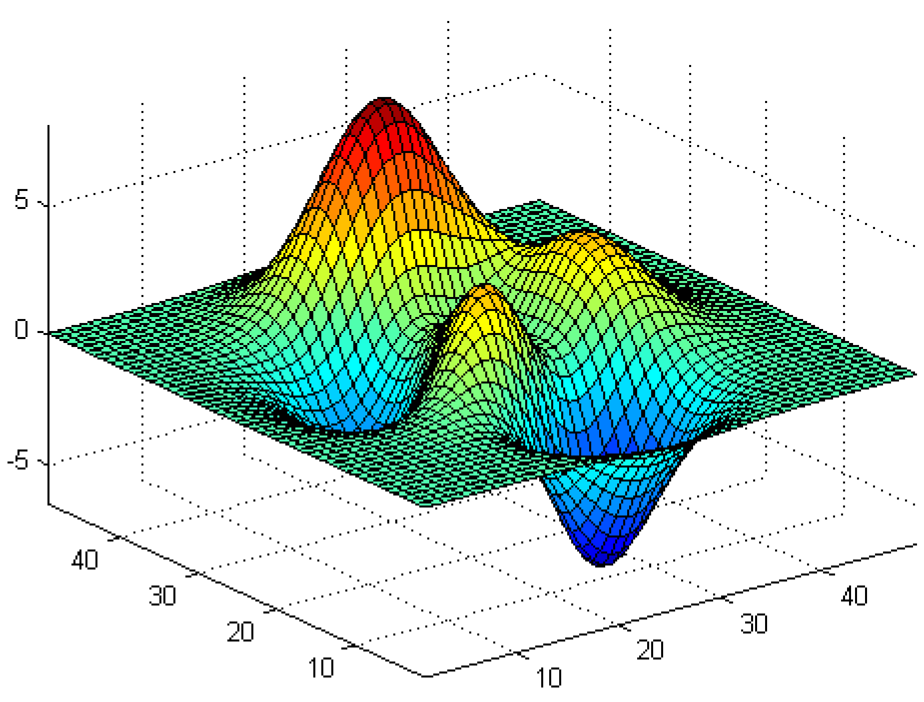
Setting the figure `Renderer` property to `zbuffer` or `painters` works around limitations of using `getframe` with the `OpenGL` renderer on some Windows systems.

Examples

Make the `peaks` function vibrate.

```
Z = peaks;
figure('Renderer','zbuffer');
surf(Z);
axis tight;
set(gca,'NextPlot','replaceChildren');
for j = 1:20
    surf(sin(2*pi*j/20)*Z,Z)
    F(j) = getframe;
end
movie(F,20) % Play the movie twenty times
```

The fifth frame of the sequence looks like this.



See Also

`axis` | `frame2im` | `gcf` | `image` | `im2frame` | `movie`

GetFullMatrix

Purpose

Matrix from Automation server workspace

Syntax

MATLAB Client

```
[xreal ximag] =
h.GetFullMatrix('varname','workspace',zreal,
   zimag)
[xreal ximag] =
GetFullMatrix(h,'varname','workspace',zreal,
   zimag)
```

IDL Method Signature

```
GetFullMatrix([in] BSTR varname, [in] BSTR
workspace, [in, out] SAFEARRAY(double) *pr, [in,
out] SAFEARRAY(double) *pi)
```

Microsoft Visual Basic Client

```
GetFullMatrix(varname As String, workspace As String,
[out] XReal As Double, [out] XImag As Double
```

Description

`[xreal ximag] = h.GetFullMatrix('varname','`*`workspace`*`',zreal,zimag)` gets matrix stored in variable `varname` from the specified `workspace` of the server attached to handle `h`. The function returns the real part in `xreal` and the imaginary part in `ximag`. The values for *`workspace`* are `base` or `global`.

`[xreal ximag] = GetFullMatrix(h,'varname','`*`workspace`*`',zreal,zimag)` is an alternate syntax.

The `zreal` and `zimag` arguments are matrices of the same size as the real and imaginary matrices (`xreal` and `ximag`) returned from the server. The `zreal` and `zimag` matrices are commonly set to zero.

Use `GetFullMatrix` for values of type `double` only. Use `GetVariable` or `GetWorkspaceData` for other types.

For VBScript clients, use the `GetWorkspaceData` and `PutWorkspaceData` functions to pass numeric data to and from the MATLAB workspace. These functions use the `variant` data type instead of the `safearray` data type used by `GetFullMatrix` and `PutFullMatrix`. VBScript does not support `safearray`.

Examples

This example uses a MATLAB client to read data from a MATLAB Automation server:

```
%Create the MATLAB server
h = actxserver('matlab.application');
%Create variable M in the base workspace of the server
h.PutFullMatrix('M','base',rand(5),zeros(5));
MReal = h.GetFullMatrix('M','base',zeros(5),zeros(5))
```

This example uses a Visual Basic .NET client to read data from a MATLAB Automation server:

1. Create the Visual Basic application. Use the `MsgBox` command to control flow between MATLAB and the application.

```
Dim MatLab As Object
Dim Result As String
Dim XReal(4, 4) As Double
Dim XImag(4, 4) As Double
Dim i, j As Integer

MatLab = CreateObject("matlab.application")
Result = MatLab.Execute("M = rand(5);")
MsgBox("In MATLAB, type" & vbCrLf _
    & "M(3,4)")
```

2. Open the MATLAB window and type:

```
M(3,4)
```

3. Click **Ok**.

4. In the Visual Basic application:

```
MatLab.GetFullMatrix("M", "base", XReal, XImag)
i = 2   %0-based array
j = 3

MsgBox("XReal(" & i + 1 & "," & j + 1 & ")" & _
    " = " & XReal(i, j))
```

5. Click **Ok** to close and terminate MATLAB.

See Also

`PutFullMatrix` | `GetVariable` | `GetWorkspaceData` | `Execute`

How To

- “Exchanging Data with the Server”
- “MATLAB COM Automation Server Interface”

getpixelposition

Purpose

Get component position in pixels

Syntax

```
position = getpixelposition(handle)
position = getpixelposition(handle,recursive)
```

Description

`position = getpixelposition(handle)` gets the position, in pixel units, of the component with handle `handle`. The `position` is returned as a four-element vector that specifies the location and size of the component: [distance from left, distance from bottom, width, height].

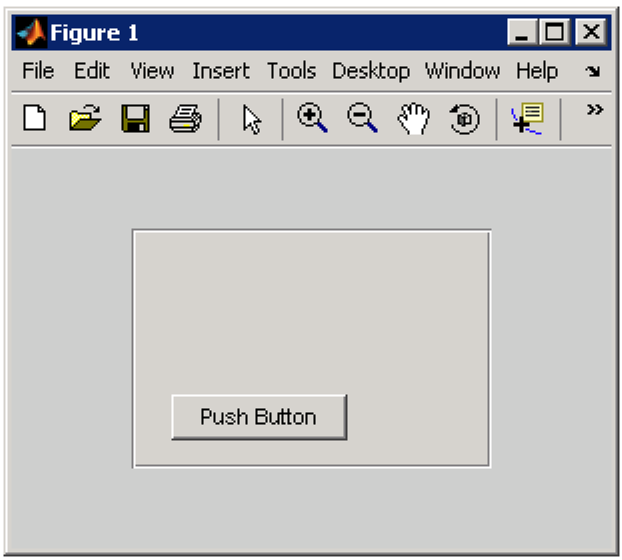
`position = getpixelposition(handle,recursive)` gets the position as above. If `recursive` is true, the returned position is relative to the parent figure of `handle`.

Use the `getpixelposition` function only to obtain coordinates for children of figures and container components (uipanels, or uibuttongroups). Results are not reliable for children of axes or other Handle Graphics objects.

Examples

This example creates a push button within a panel, and then retrieves its position, in pixels, relative to the panel.

```
f = figure('Position',[300 300 300 200]);
p = uipanel('Position',[.2 .2 .6 .6]);
h1 = uicontrol(p,'Style','PushButton','Units','Normalized',...
               'String','Push Button','Position',[.1 .1 .5 .2]);
pos1 = getpixelposition(h1)

pos1 =
     18.6000   12.6000   88.0000   23.2000
```



The following statement retrieves the position of the push button, in pixels, relative to the figure.

```
pos1 = getpixelposition(h1,true)

pos1 =
      79.6000   53.6000   88.0000   23.2000
```

See Also

`setpixelposition` | `uicontrol` | `uipanel`

Purpose

Preference

Syntax

```
getpref('group','pref')
getpref('group','pref',default)
getpref('group',{'pref1','pref2',...'prefn'})
getpref('group',{'pref1',...'prefn'},{default1,...defaultn})
getpref('group')
getpref
```

Description

`getpref('group','pref')` returns the value for the preference specified by `group` and `pref`. It is an error to get a preference that does not exist.

`group` labels a related collection of preferences. You can choose any name that is a legal variable name, and is descriptive enough to be unique, e.g. `'ApplicationOnePrefs'`. The input argument `pref` identifies an individual preference in that group, and must be a legal variable name.

`getpref('group','pref',default)` returns the current value if the preference specified by `group` and `pref` exists. Otherwise creates the preference with the specified default value and returns that value.

`getpref('group',{'pref1','pref2',...'prefn'})` returns a cell array containing the values for the preferences specified by `group` and the cell array of preference names. The return value is the same size as the input cell array. It is an error if any of the preferences do not exist.

`getpref('group',{'pref1',...'prefn'},{default1,...defaultn})` returns a cell array with the current values of the preferences specified by `group` and the cell array of preference names. Any preference that does not exist is created with the specified default value and returned.

`getpref('group')` returns the names and values of all preferences in the `group` as a structure.

`getpref` returns all groups and preferences as a structure.

Note Preference values are persistent and maintain their values between MATLAB sessions. Where they are stored is system dependent.

Examples

Example 1

```
addpref('mytoolbox','version','1.0')
getpref('mytoolbox','version')

ans =
     1.0
```

Example 2

```
rmpref('mytoolbox','version')
getpref('mytoolbox','version','1.0');
getpref('mytoolbox','version')

ans =
     1.0
```

See Also

addpref | ispref | rmpref | setpref | uigetpref | uisetpref

Purpose

List profiles and file formats supported by `VideoWriter`

Syntax

```
profiles = VideoWriter.getProfiles()
```

Description

`profiles = VideoWriter.getProfiles()` returns an array of `audiovideo.writer.ProfileInfo` objects that indicate the types of files `VideoWriter` can create.

Output Arguments

profiles

An array of `audiovideo.writer.ProfileInfo` objects, which have the following read-only properties:

`Name`	String profile name, such as `'Uncompressed AVI'`.
`Description`	String description of the profile.
`FileExtensions`	Cell array of strings containing file extensions supported by the file format.
`ColorChannels`	Number of color channels in each output video frame.
`CompressionRatio`	Number greater than 1 that specifies the target ratio between the number of bytes in the input image and the number of bytes in the compressed image. Only applies to objects associated with Motion JPEG 2000 files. Default: `10`.
`FrameRate`	Rate of playback for the video in frames per second. Default: `30`.

LosslessCompression	Boolean value (logical `true` or `false`) that specifies whether to use reversible mode, so that the decompressed data is identical to the input data. When `true`, `VideoWriter` ignores values for `CompressionRatio`. Only applies to objects associated with Motion JPEG 2000 files.
MJ2BitDepth	Number of least significant bits in the input image data, from 1 to 16. Only applies to objects associated with Motion JPEG 2000 files.
Quality	Number from 0 through 100. Higher values correspond to higher quality video and larger files. Only applies to objects associated with the `MPEG-4` or `Motion JPEG AVI` profile. Default: `75`.
VideoBitsPerPixel	Number of bits per pixel in each output video frame.
VideoCompressionMethod	String indicating the type of video compression, such as `'None'` or `'Motion JPEG'`.
VideoFormat	String indicating the MATLAB representation of the video format, such as `'RGB24'`.

Examples

Profile Information

View the list of available profiles and specific information about the `'Uncompressed AVI'` profile.

```
profiles = VideoWriter.getProfiles()

uncompAVI = find(ismember({profiles.Name},'Uncompressed AVI'));
profiles(uncompAVI)
```

```
profiles(uncompAVI).FileExtensions
```

See Also

`VideoWriter`

getReport (MException)

Purpose

Get error message for exception

Syntax

```
msgString = getReport(exception)
msgString = getReport(exception, type)
msgString = getReport(exception, type, 'hyperlinks', value)
```

Description

msgString = getReport(*exception*) returns a formatted message string, *msgString*, from an exception. This exception is represented by the *exception* input which is a scalar object of the MException class. The message string returned by getReport is the same as the error message displayed by MATLAB when it throws this exception.

msgString = getReport(*exception*, *type*) returns a message string that either describes just the error (**basic** type), or shows the error and the stack as well (**extended** type). The *type* argument, when used, must be the second argument in the input argument list. See "Examples" on page 1-2001 , below.

type Option	Displayed Text
'extended'	Display line number, error message, and cause and stack summary (default)
'basic'	Display line number and error message

msgString = getReport(*exception*, *type*, '**hyperlinks**', *value*) returns a message string that either does or does not include active hyperlinks to the failing lines in the code. See the table below for the valid choices for *value*. The **hyperlinks** and *value* arguments, when used, must be the third and fourth arguments in the input argument list.

`value` Option	Action
'on'	Display hyperlinks to failing lines (default)
'off'	Do not display hyperlinks to failing lines
'default'	Use the default for the Command Window to determine whether or not to use hyperlinks in the error message

Examples

This function attempts to read from a file that does not exist. When you call it, pass either `basic' or `extended' to specify the type of report you want to see displayed:

```
function line = test_getreport(file, rpttype)
try
    line = read_file(file);
catch exc
    getReport(exc, rpttype)
end

function line = read_file(file)
    fid = fopen(file, 'r');
    line = fread(fid);
```

The basic option displays only the error message:

```
test_getreport('filethatdoesnotexist.txt', 'basic')
ans =

Error using fread

Invalid file identifier.  Use fopen to generate a valid file identifie
```

The extended option displays the error message and error call stack:

```
test_getreport('filethatdoesnotexist.txt', 'extended')
ans =
```

```
Error using fread

Invalid file identifier.  Use fopen to generate a valid file identifier.

Error in test_getreport>read_file (line 10)

    line = fread(fid);

Error in test_getreport (line 3)

    line = read_file(file);
```

See Also

try | catch | error | assert | MException | throw(MException) | rethrow(MException) | throwAsCaller(MException) | addCause(MException) | last(MException)

Purpose Extract data samples into new `tscollection` object

Syntax

```
tsc2 = getsampleusingtime(tsc1,Time)
tsc2 = getsampleusingtime(tsc1,StartTime,EndTime)
```

Description `tsc2 = getsampleusingtime(tsc1,Time)` returns a new `tscollection` tsc2 with a single sample corresponding to Time in tsc1.

`tsc2 = getsampleusingtime(tsc1,StartTime,EndTime)` returns a new `tscollection` tsc2 with samples between the times StartTime and EndTime in tsc1.

Tips When the time vector in ts1 is numeric, `StartTime` and `EndTime` must also be numeric. When the times in ts1 are date strings and the `StartTime` and `EndTime` values are numeric, then the `StartTime` and `EndTime` values are treated as `datenum` values.

See Also `tscollection`

Tiff.getTag

Purpose

Value of specified tag

Syntax

```
tagValue = getTag(tagId)
```

Description

`tagValue = getTag(tagId)` retrieves the value of the TIFF tag specified by `tagId`. You can specify `tagId` as a character string (`'ImageWidth'`) or using the numeric tag identifier defined by the TIFF specification (256). To see a list of all the tags with their numeric identifiers, view the value of the Tiff object `TagID` property. Use the `TagID` property to specify the value of a tag. For example, `Tiff.TagID.ImageWidth` is equivalent to the tag's numeric identifier.

Examples

Open a `Tiff` object, and get the value of a tag. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path:

```
t = Tiff('myfile.tif', 'r');
% Specify tag by name.
tagval = t.getTag('ImageWidth');
%
% Specify tag by numeric identifier.
tagval1 = t.getTag(256);
%
% Specify tag by name.
tagval2 = t.getTag('t.TagID.ImageWidth');
```

References

This method corresponds to the `TIFFGetField` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTiff - TIFF Library and Utilities`.

See Also

`Tiff.setTag`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

Purpose List of recognized TIFF tags

Syntax

```
tagNames = Tiff.getTagNames()
```

Description `tagNames = Tiff.getTagNames()` returns a cell array of TIFF tags recognized by the `Tiff` object.

Examples Retrieve a list of TIFF tags recognized by the `Tiff` object.

```
Tiff.getTagNames

ans =

    'SubFileType'
    'ImageWidth'
    'ImageLength'
    'BitsPerSample'
    'Compression'
    'Photometric'
    'Thresholding'
    'FillOrder'
    'DocumentName'
    'ImageDescription'
     .
     .
     .
```

See Also `Tiff.getTag`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

gettimeseriesnames

Purpose

Cell array of names of `timeseries` objects in `tscollection` object

Syntax

```
names = gettimeseriesnames(tsc)
```

Description

`names = gettimeseriesnames(tsc)` returns `names` of `timeseries` objects in a `tscollection` object `tsc`. `names` is a cell array of strings.

Examples

1 Create `timeseries` objects `a` and `b`.

```
a = timeseries(rand(1000,1),'name','position');
b = timeseries(rand(1000,1),'name','response');
```

2 Create a `tscollection` object that includes these two time series.

```
tsc = tscollection({a,b});
```

3 Get the names of the `timeseries` objects in `tsc`.

```
names = gettimeseriesnames(tsc)

names =

    'position'    'response'
```

See Also

`timeseries` | `tscollection`

Purpose

New `timeseries` object with samples occurring at or after event

Syntax

```
ts1 = gettsafteratevent(ts,event)
ts1 = gettsafteratevent(ts,event,n)
```

Description

`ts1 = gettsafteratevent(ts,event)` returns a new `timeseries` object ts1 with samples occurring at and after an event in ts, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of the time series ts that matches the event name specifies the time.

`ts1 = gettsafteratevent(ts,event,n)` returns a new `timeseries` object ts1 with samples at and after an event in ts, where n is the number of the event occurrence with a matching event name.

Tips

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent` | `gettsbeforeevent` | `gettsbetweenevents` | `tsdata.event` | `timeseries`

gettsafterevent

Purpose

New `timeseries` object with samples occurring after event

Syntax

```
ts1 = gettsafterevent(ts,event)
ts1 = ttsafterevent(ts,event,n)
```

Description

`ts1 = gettsafterevent(ts,event)` returns a new `timeseries` object ts1 with samples occurring after an event in ts, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of `ts` that matches the event name specifies the time.

`ts1 = ttsafterevent(ts,event,n)` returns a new `timeseries` object `ts1` with samples occurring after an event in time series `ts`, where n is the number of the event occurrence with a matching event name.

Tips

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafteratevent` | `gettsbeforeevent` | `gettsbetweenevents` | `timeseries` | `tsdata.event`

Purpose

New `timeseries` object with samples occurring at event

Syntax

```
ts1 = gettsatevent(ts,event)
ts1 = gettsatevent(ts,event,n)
```

Description

`ts1 = gettsatevent(ts,event)` returns a new `timeseries` object `ts1` with samples occurring at an event in `ts`, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of `ts` that matches the event name specifies the time.

`ts1 = gettsatevent(ts,event,n)` returns a new time series `ts1` with samples occurring at an event in time series `ts`, where n is the number of the event occurrence with a matching event name.

Tips

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in the `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent` | `gettsafteratevent` | `gettsbeforeevent` | `gettsbetweenevents` | `timeseries`

gettsbeforeatevent

Purpose

New `timeseries` object with samples occurring before or at event

Syntax

```
ts1 = gettsbeforeatevent(ts,event)
ts1 = gettsbeforeatevent(ts,event,n)
```

Description

`ts1 = gettsbeforeatevent(ts,event)` returns a new `timeseries` object `ts1` with samples occurring at and before an event in `ts`, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of `ts` that matches the event name specifies the time.

`ts1 = gettsbeforeatevent(ts,event,n)` returns a new `timeseries` object `ts1` with samples occurring at and before an event in time series `ts`, where n is the number of the event occurrence with a matching event name.

Tips

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent` | `gettsbeforeevent` | `gettsbetweenevents` | `tsdata.event`

gettsbeforeevent

Purpose

New `timeseries` object with samples occurring before event

Syntax

```
ts1 = gettsbeforeevent(ts,event)
ts1 = gettsbeforeevent(ts,event,n)
```

Description

`ts1 = gettsbeforeevent(ts,event)` returns a new `timeseries` object `ts1` with samples occurring before an event in `ts`, where event can be either a `tsdata.event` object or a string. When event is a `tsdata.event` object, the time defined by event is used. When event is a string, the first `tsdata.event` object in the `Events` property of `ts` that matches the event name specifies the time.

`ts1 = gettsbeforeevent(ts,event,n)` returns a new `timeseries` object `ts1` with samples occurring before an event in `ts`, where n is the number of the event occurrence with a matching event name.

Tips

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent` | `gettsbeforeatevent` | `gettsbetweenevents` | `tsdata.event`

gettsbetweenevents

Purpose

New `timeseries` object with samples occurring between events

Syntax

```
ts1 = gettsbetweenevents(ts,event1,event2)
ts1 = gettsbetweenevents(ts,event1,event2,n1,n2)
```

Description

`ts1 = gettsbetweenevents(ts,event1,event2)` returns a new `timeseries` object ts1 with samples occurring between events in `ts`, where `event1` and `event2` can be either a `tsdata.event` object or a string. When `event1` and `event2` are `tsdata.event` objects, the time defined by the events is used. When `event1` and `event2` are strings, the first `tsdata.event` object in the `Events` property of `ts` that matches the event names specifies the time.

`ts1 = gettsbetweenevents(ts,event1,event2,n1,n2)` returns a new `timeseries` object ts1 with samples occurring between events in `ts`, where `n1` and `n2` are the `n`th occurrences of the events with matching event names.

Tips

When the `timeseries` object `ts` contains date strings and `event` uses numeric time, the time selected by the `event` is treated as a date that is calculated relative to the `StartDate` property in `ts.TimeInfo`.

When `ts` uses numeric time and `event` uses calendar dates, the time selected by the `event` is treated as a numeric value that is not associated with a calendar date.

See Also

`gettsafterevent` | `gettsbeforeevent` | `tsdata.event`

Purpose

Data from variable in Automation server workspace

Syntax

MATLAB Client

```
D = h.GetVariable('varname','workspace')
D = GetVariable(h,'varname','workspace')
```

IDL Method Signature

```
HRESULT GetVariable([in] BSTR varname, [in] BSTR
workspace, [out, retval] VARIANT* pdata)
```

Microsoft Visual Basic Client

```
GetVariable(varname As String, workspace As
String) As Object
```

Description

`D = h.GetVariable('varname','workspace')` gets data stored in variable `varname` from the specified `workspace` of the server attached to handle `h` and returns it in output argument `D`. The values for *`workspace`* are `base` or `global`.

`D = GetVariable(h,'varname','workspace')` is an alternate syntax.

Do *not* use `GetVariable` on sparse arrays, structures, or function handles.

If your scripting language requires a result be returned explicitly, use the `GetVariable` function in place of `GetWorkspaceData`, `GetFullMatrix` or `GetCharArray`.

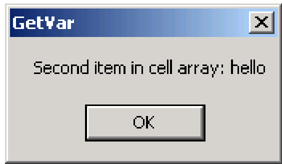
Examples

This example uses a MATLAB client to read data from a MATLAB Automation server:

```
%Create the MATLAB server
h = actxserver('matlab.application');
%Create variable C1 in the base workspace of the server
h.PutWorkspaceData('C1', 'base', {25.72, 'hello', rand(4)});
%The client reads the data
C2 = h.GetVariable('C1','base')
```

This example uses a Visual Basic .NET client to read data from a MATLAB Automation server:

```
Dim Matlab As Object
Dim Result As String
Dim C2 As Variant
Matlab = CreateObject("matlab.application")
Result = Matlab.Execute("C1 = {25.72, 'hello', rand(4)};")
C2 = Matlab.GetVariable("C1", "base")
MsgBox("Second item in cell array: " & C2(0, 1))
```



See Also

`GetWorkspaceData` | `GetFullMatrix` | `GetCharArray` | `Execute`

How To

- “MATLAB COM Automation Server Interface”
- “Exchanging Data with the Server”

Purpose

LibTIFF library version

Syntax

```
versionString = Tiff.getVersion()
```

Description

`versionString = Tiff.getVersion()` returns the version number and other information about the LibTIFF library.

Examples

Display version of LibTIFF library:

```
Tiff.getVersion

ans =

LIBTIFF, Version 3.9.5
Copyright (c) 1988-1996 Sam Leffler
Copyright (c) 1991-1996 Silicon Graphics, Inc.
```

References

This method corresponds to the `TIFFGetVersion` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

GetWorkspaceData

Purpose

Data from Automation server workspace

Syntax

MATLAB Client

```
D = h.GetWorkspaceData('varname','workspace')
D = GetWorkspaceData(h,'varname','workspace')
```

IDL Method Signature

```
HRESULT GetWorkspaceData([in] BSTR varname, [in] BSTR
workspace, [out] VARIANT* pdata)
```

Microsoft Visual Basic Client

```
GetWorkspaceData(varname As String, workspace
As String) As Object
```

Description

`D = h.GetWorkspaceData('varname','workspace')` gets data stored in variable `varname` from the specified `workspace` of the server attached to handle `h` and returns it in output argument `D`. The values for *`workspace`* are `base` or `global`.

`D = GetWorkspaceData(h,'varname','workspace')` is an alternate syntax.

Use `GetWorkspaceData` instead of `GetFullMatrix` and `GetCharArray` to get numeric and character array data, respectively. Do *not* use `GetWorkspaceData` on sparse arrays, structures, or function handles.

For VBScript clients, use the `GetWorkspaceData` and `PutWorkspaceData` functions to pass numeric data to and from the MATLAB workspace. These functions use the `variant` data type instead of the `safearray` data type used by `GetFullMatrix` and `PutFullMatrix`. VBScript does not support `safearray`.

Examples

This example uses a MATLAB client to read data from a MATLAB Automation server:

```
%Create the MATLAB server
h = actxserver('matlab.application');
%Create cell array C1 in the base workspace of the server
```

```
h.PutWorkspaceData('C1', 'base', ...
    {25.72, 'hello', rand(4)});
C2 = h.GetWorkspaceData('C1', 'base')
```

This example uses a Visual Basic .NET client to read data from a MATLAB Automation server:

```
Dim Matlab As Object
Dim C2 As Variant
Dim Result As String
Matlab = CreateObject("matlab.application")
Result = MatLab.Execute("C1 = {25.72, 'hello', rand(4)};")
MsgBox("In MATLAB, type" & vbCrLf & "C1")
Matlab.GetWorkspaceData("C1", "base", C2)
MsgBox("second value of C1 = " & C2(0, 1))
```

See Also

`PutWorkspaceData` | `GetFullMatrix` | `GetCharArray` | `GetVariable` | `Execute`

How To

- “MATLAB COM Automation Server Interface”
- “Exchanging Data with the Server”

ginput

Purpose

Graphical input from mouse or cursor

Syntax

```
[x,y] = ginput(n)
[x,y] = ginput
[x,y,button] = ginput(...)
```

Description

`ginput` raises crosshairs in the current axes to for you to identify points in the figure, positioning the cursor with the mouse. The figure must have focus before `ginput` can receive input. If it has no axes, one is created upon the first click or keypress.

`[x,y] = ginput(n)` enables you to identify `n` points from the current axes and returns their *x*- and *y*-coordinates in the `x` and `y` column vectors. Press the **Return** key to terminate the input before entering `n` points.

`[x,y] = ginput` gathers an unlimited number of points until you press the **Return** key.

`[x,y,button] = ginput(...)` returns the *x*-coordinates, the *y*-coordinates, and the button or key designation. `button` is a vector of integers indicating which mouse buttons you pressed (1 for left, 2 for middle, 3 for right), or ASCII numbers indicating which keys on the keyboard you pressed.

Clicking an axes makes that axes the current axes. Even if you set the current axes before calling `ginput`, whichever axes you click becomes the current axes and `ginput` returns points relative to that axes. If you select points from multiple axes, the results returned are relative to the coordinate system of the axes they come from.

Note MATLAB returns errors such as the following if you start MATLAB with the `noFigureWindows` or `nodisplay` flag and then run `ginput`:

```
Error using ginput (line 31)
Terminal mode is no longer supported
```

Definitions

Coordinates returned by `ginput` are scaled to the `XLim` and `YLim` bounds of the axes you click (data units). Setting the axes or figure `Units` property has no effect on the output from `ginput`. You can click anywhere within the figure canvas to obtain coordinates. If you click outside the axes limits, `ginput` extrapolates coordinate values so they are still relative to the axes origin.

The figure `CurrentPoint` property, by contrast, is always returned in figure Units, irrespective of axes Units or limits.

Examples

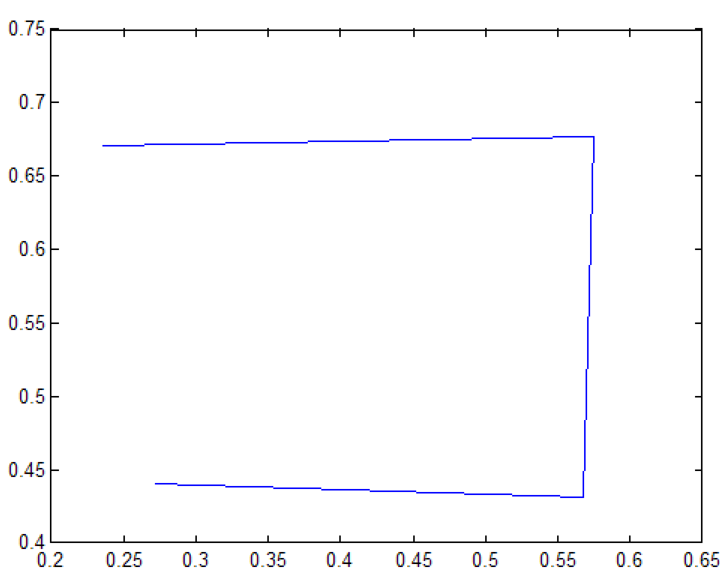
Pick 4 two-dimensional points from the figure window.

```
[x,y] = ginput(4)
```

Position the cursor with the mouse. Enter data points by pressing a mouse button or a key on the keyboard. To terminate input before entering 4 points, press the **Return** key.

```
x =
    0.2362
    0.5749
    0.5680
    0.2707

y =
    0.6711
    0.6769
    0.4313
```

```
    0.4401

plot(x,y)
```



In this example, `plot` rescaled the axes *x*-limits and *y*-limits from `[0 1]` and `[0 1]` to `[0.20 0.65]` and `[0.40 0.75]`. The rescaling occurred because the axes `XLimMode` and `YLimMode` are set to `'auto'` (the default). Consider setting `XLimMode` and `YLimMode` to `'manual'` if you want to maintain consistency when you gather results from `ginput` and plot them together.

See Also

`gtext`

Tutorials

- "Selecting Plotting Points from the Screen"
- "Subsetting a Graphics Image (Cropping)"

Purpose

Declare global variables

Syntax

`global X Y Z`

Description

`global X Y Z` defines `X`, `Y`, and `Z` as global in scope.

Ordinarily, each MATLAB function has its own local variables, which are separate from those of other functions, and from those of the base workspace. However, if several functions, and possibly the base workspace, all declare a particular name as global, they all share a single copy of that variable. Any assignment to that variable, in any function, is available to all the functions declaring it global.

If the global variable does not exist the first time you issue the `global` statement, it is initialized to the empty matrix.

If a variable with the same name as the global variable already exists in the current workspace, MATLAB issues a warning and changes the value of that variable to match the global.

Tips

Use `clear global` *`variable`* to clear a global variable from the global workspace. Use `clear` *`variable`* to clear the global link from the current workspace without affecting the value of the global.

To use a global within a callback, declare the global, use it, then clear the global link from the workspace. This avoids declaring the global after it has been referenced. For example,

```
cbstr = sprintf('%s, %s, %s, %s, %s', ...
   'global MY_GLOBAL',                 ...
   'MY_GLOBAL = 100',                  ...
   'disp(MY_GLOBAL)',                  ...
   'MY_GLOBAL = MY_GLOBAL+1',          ...
   'clear MY_GLOBAL');

uicontrol('style', 'pushbutton', 'CallBack', cbstr, ...
   'string', 'count')
```

There is no function form of the global command (i.e., you cannot use parentheses and quote the variable names).

Examples

Example 1

Type `edit global_demo1` at the command line and enter the following function definition in a new file named `global_demo1.m`.

```
function global_demo1(num)
global globalvar

if nargin > 0
    globalvar = num;
end

fprintf('Global variable in function %s is %d\n', ...
    mfilename, globalvar);
```

This function declares a global variable named `globalvar`. If you pass a value when calling the function, the function stores the value in `globalvar` and then displays it. If you call the function with no arguments, the function just displays the value last written to the global workspace.

Create another function `global_demo2` just like it. These two functions have separate function workspaces, but share a common global workspace:

```
copyfile global_demo1.m global_demo2.m
```

Call `global_demo1`, passing a numeric value. Then call `global_demo2` with no value. You can see that the latter function has global access to the value that was passed to `global_demo1`.

```
global_demo1(1357);
Global variable in function global_demo1 is 1357

global_demo2
Global variable in function global_demo2 is 1357
```

Now set the value in `global_demo2` and read it in `global_demo1`:

```
global_demo2(2468)
Global variable in function global_demo2 is 2468

global_demo1
Global variable in function global_demo1 is 2468
```

Example 2

Call the function `global_demo1` that was defined in the previous example to assign a value to variable `globalvar` in the global workspace. Even though the variable is global, it is not accessible outside of the function workspace:

```
clear all
global_demo1(1357);
Global variable in function global_demo1 is 1357

if exist('globalvar', 'var')
    fprintf('Global variable is set to %d\n', globalvar);
else
    fprintf('Global variable is not available at the command line.\n')
end

Global variable is not available at the command line.
```

Now declare `globalvar` as a global variable at the MATLAB command line. Run the same statements to display the variable and this time you can see that the value assigned by the function is also available as a global variable in the base workspace:

```
global globalvar

if exist('globalvar', 'var')
    fprintf('Global variable is set to %d\n', globalvar);
else
```

```
    fprintf('Global variable is not available at the command line.\n');
end

Global variable is set to 1357
```

Example 3

Here is the code for the functions `tic` and `toc` (some comments abridged). These functions manipulate a stopwatch-like timer. The global variable `TICTOC` is shared by the two functions, but it is invisible in the base workspace or in any other functions that do not declare it.

```
function tic
%    TIC Start a stopwatch timer.
%        TIC; any stuff; TOC
%    prints the time required.
%    See also: TOC, CLOCK.
global TICTOC
TICTOC = clock;

function t = toc
%    TOC Read the stopwatch timer.
%    TOC prints the elapsed time since TIC was used.
%    t = TOC; saves elapsed time in t, does not print.
%    See also: TIC, ETIME.
global TICTOC
if nargout < 1
    elapsed_time = etime(clock, TICTOC)
else
    t = etime(clock, TICTOC);
end
```

See Also

`clear` | `isglobal` | `who`

Purpose

Generalized minimum residual method (with restarts)

Syntax

```
x = gmres(A,b)
gmres(A,b,restart)
gmres(A,b,restart,tol)
gmres(A,b,restart,tol,maxit)
gmres(A,b,restart,tol,maxit,M)
gmres(A,b,restart,tol,maxit,M1,M2)
gmres(A,b,restart,tol,maxit,M1,M2,x0)
[x,flag] = gmres(A,b,...)
[x,flag,relres] = gmres(A,b,...)
[x,flag,relres,iter] = gmres(A,b,...)
[x,flag,relres,iter,resvec] = gmres(A,b,...)
```

Description

`x = gmres(A,b)` attempts to solve the system of linear equations `A*x = b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. `A` can be a function handle, `afun`, such that `afun(x)` returns `A*x`. For this syntax, `gmres` does not restart; the maximum number of iterations is `min(n,10)`.

"Parameterizing Functions" explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `gmres` converges, a message to that effect is displayed. If `gmres` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`gmres(A,b,restart)` restarts the method every `restart` inner iterations. The maximum number of outer iterations is `min(n/restart,10)`. The maximum number of total iterations is `restart*min(n/restart,10)`. If `restart` is `n` or `[]`, then `gmres` does not restart and the maximum number of total iterations is `min(n,10)`.

`gmres(A,b,restart,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `gmres` uses the default, `1e-6`.

`gmres(A,b,restart,tol,maxit)` specifies the maximum number of outer iterations, i.e., the total number of iterations does not exceed `restart*maxit`. If `maxit` is `[]` then `gmres` uses the default, `min(n/restart,10)`. If `restart` is `n` or `[]`, then the maximum number of total iterations is `maxit` (instead of `restart*maxit`).

`gmres(A,b,restart,tol,maxit,M)` and `gmres(A,b,restart,tol,maxit,M1,M2)` use preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `gmres` applies no preconditioner. `M` can be a function handle `mfun` such that `mfun(x)` returns `M\x`.

`gmres(A,b,restart,tol,maxit,M1,M2,x0)` specifies the first initial guess. If `x0` is `[]`, then `gmres` uses the default, an all-zero vector.

`[x,flag] = gmres(A,b,...)` also returns a convergence flag:

`flag = 0`	`gmres` converged to the desired tolerance `tol` within `maxit` outer iterations.
`flag = 1`	`gmres` iterated `maxit` times but did not converge.
`flag = 2`	Preconditioner `M` was ill-conditioned.
`flag = 3`	`gmres` stagnated. (Two consecutive iterates were the same.)

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = gmres(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`. The third output, `relres`, is the relative residual of the preconditioned system.

`[x,flag,relres,iter] = gmres(A,b,...)` also returns both the outer and inner iteration numbers at which `x` was computed, where `0 <= iter(1) <= maxit` and `0 <= iter(2) <= restart`.

`[x,flag,relres,iter,resvec] = gmres(A,b,...)` also returns a vector of the residual norms at each inner iteration. These are the residual norms for the preconditioned system.

Examples

Using gmres with a Matrix Input

```
A = gallery('wilk',21);
b = sum(A,2);
tol = 1e-12;
maxit = 15;
M1 = diag([10:-1:1 1 1:10]);

x = gmres(A,b,10,tol,maxit,M1);
```

displays the following message:

```
gmres(10) converged at outer iteration 2 (inner iteration 9) to
a solution with relative residual 3.3e-013
```

Using gmres with a Function Handle

This example replaces the matrix `A` in the previous example with a handle to a matrix-vector product function `afun`, and the preconditioner `M1` with a handle to a backsolve function `mfun`. The example is contained in a function `run_gmres` that

- Calls `gmres` with the function handle `@afun` as its first argument.
- Contains `afun` and `mfun` as nested functions, so that all variables in `run_gmres` are available to `afun` and `mfun`.

The following shows the code for `run_gmres`:

```
function x1 = run_gmres
n = 21;
b = afun(ones(n,1));
tol = 1e-12;  maxit = 15;
x1 = gmres(@afun,b,10,tol,maxit,@mfun);

    function y = afun(x)
        y = [0; x(1:n-1)] + ...
              [((n-1)/2:-1:0)'; (1:(n-1)/2)'].*x + ...
              [x(2:n); 0];
    end
```

```
    function y = mfun(r)
        y = r ./ [((n-1)/2:-1:1)'; 1; (1:(n-1)/2)'];
    end
end
```

When you enter

```
x1 = run_gmres;
```

MATLAB software displays the message

```
gmres(10) converged at outer iteration 2 (inner iteration 10)
to a solution with relative residual 1.1e-013.
```

Using a Preconditioner without Restart

This example demonstrates the use of a preconditioner without restarting `gmres`.

1

Load `west0479`, a real 479-by-479 nonsymmetric sparse matrix:

```
load west0479;
A = west0479;
```

2

Set the tolerance and maximum number of iterations:

```
tol = 1e-12; maxit = 20;
```

3

Define `b` so that the true solution is a vector of all ones.

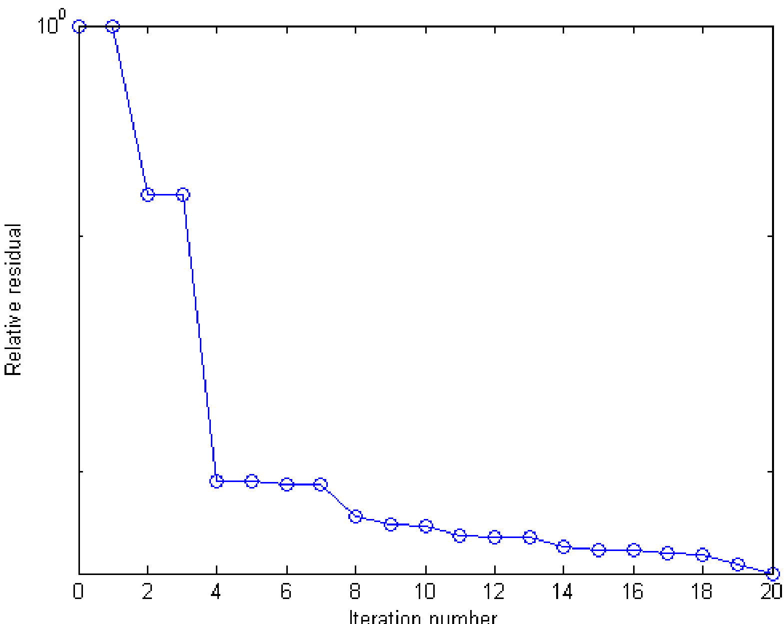
```
b = full(sum(A,2));
[x0,fl0,rr0,it0,rv0] = gmres(A,b,[],tol,maxit);
```

`fl0` is 1 because `gmres` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. The best approximate solution that `gmres` returns is the last one (as indicated by `it0(2) = 20`). MATLAB stores the residual history in `rv0`.

4

Plot the behavior of `gmres`:

```
semilogy(0:maxit,rv0/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



The plot shows that the solution converges slowly. A preconditioner may improve the outcome.

5

Use `ilu` to form the preconditioner, since `A` is nonsymmetric :

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));
Error using ilu
There is a pivot equal to zero. Consider decreasing
the drop tolerance or consider using the 'udiag' option.
```

Note MATLAB cannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

6

As indicated by the error message, try again with a reduced drop tolerance:

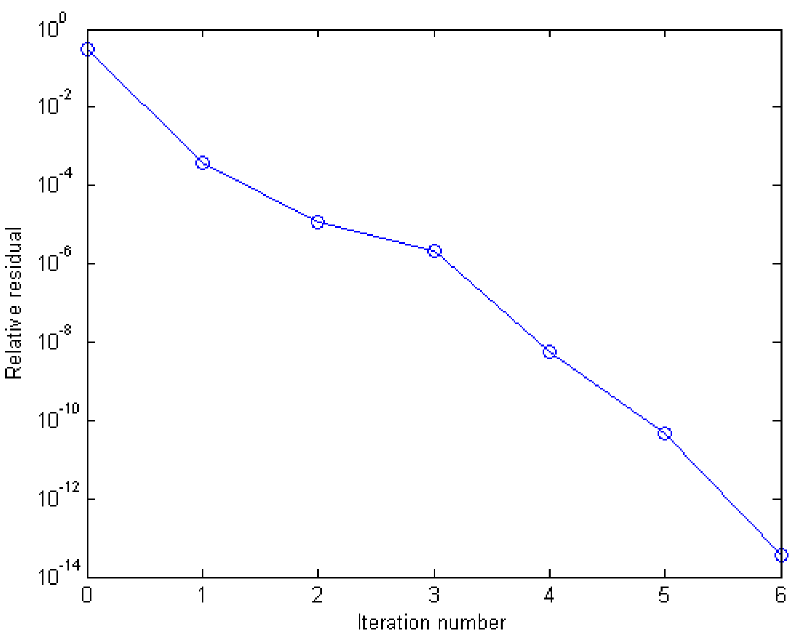
```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = gmres(A,b,[],tol,maxit,L,U);
```

`fl1` is 0 because `gmres` drives the relative residual to `9.5436e-14` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the sixth iteration (the value of `it1(2)`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output, `rv1(1)`, is `norm(M\b)`, where `M = L*U`. The output, `rv1(7)`, is `norm(U\(L\(b-A*x1)))`.

7

Follow the progress of `gmres` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0):

```
semilogy(0:it1(2),rv1/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



Using a Preconditioner with Restart

This example demonstrates the use of a preconditioner with restarted `gmres`.

1

Load `west0479`, a real 479-by-479 nonsymmetric sparse matrix:

```
load west0479;
A = west0479;
```

2

Define `b` so that the true solution is a vector of all ones:

```
b = full(sum(A,2));
```

3

Construct an incomplete LU preconditioner as in the previous example:

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
```

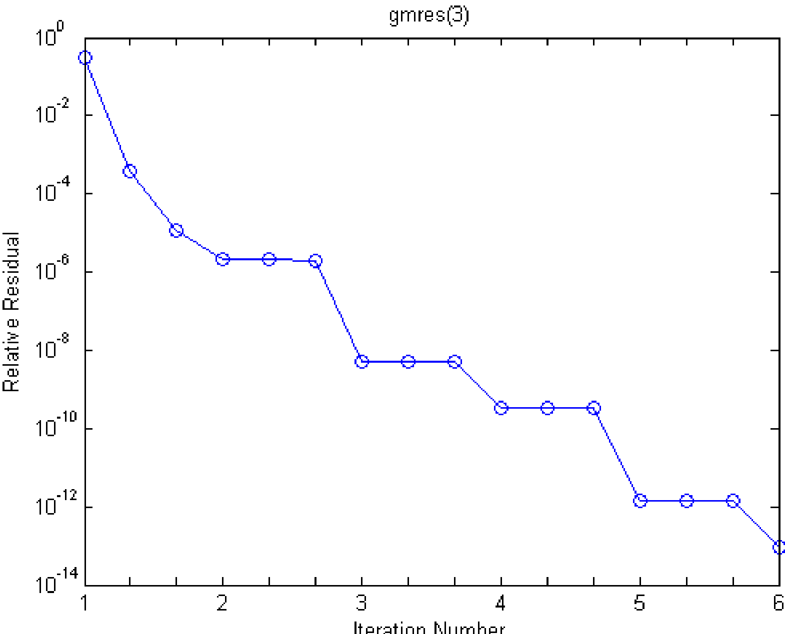
The benefit to using restarted `gmres` is to limit the amount of memory required to execute the method. Without restart, `gmres` requires `maxit` vectors of storage to keep the basis of the Krylov subspace. Also, `gmres` must orthogonalize against all of the previous vectors at each step. Restarting limits the amount of workspace used and the amount of work done per outer iteration. Note that even though preconditioned `gmres` converged in six iterations above, the algorithm allowed for as many as twenty basis vectors and therefore, allocated all of that space up front.

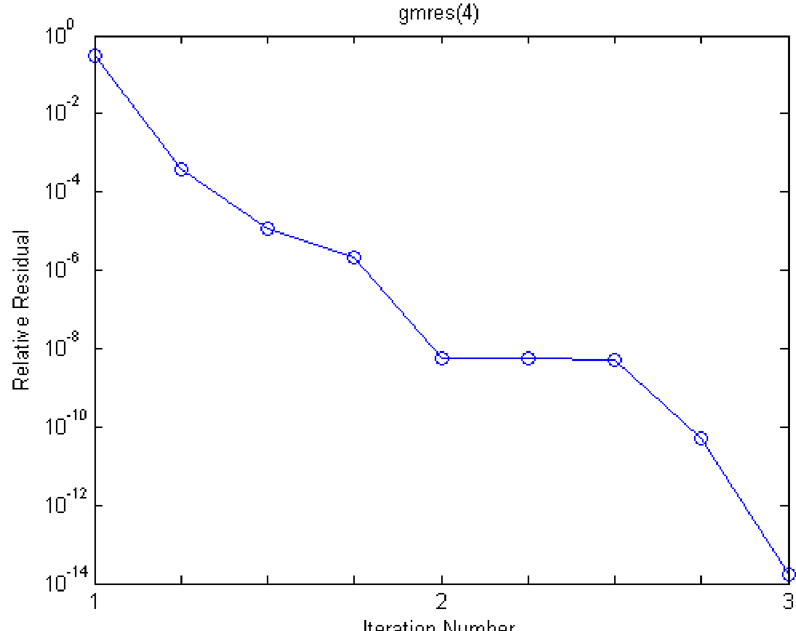
4

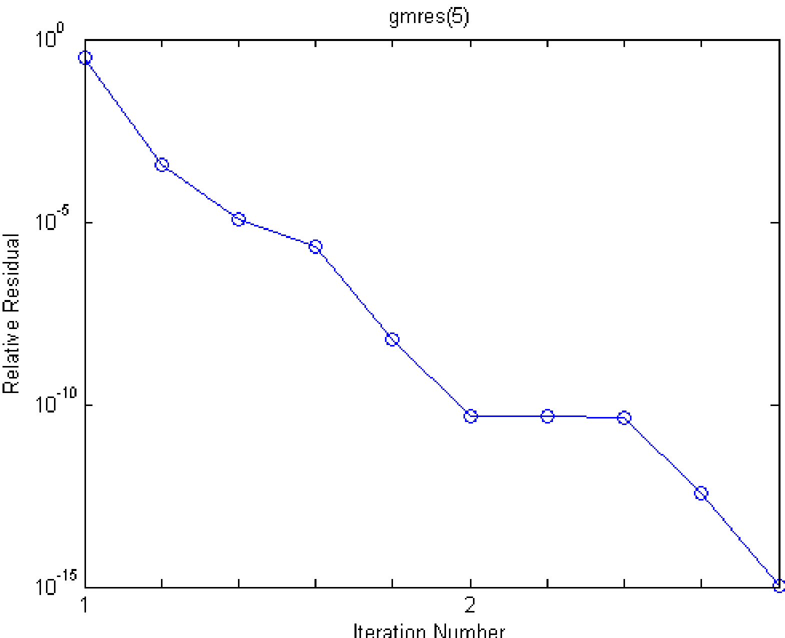
Execute `gmres(3)`, `gmres(4)`, and `gmres(5)`:

```
tol = 1e-12; maxit = 20;
re3 = 3;
[x3,fl3,rr3,it3,rv3] = gmres(A,b,re3,tol,maxit,L,U);
re4 = 4;
[x4,fl4,rr4,it4,rv4] = gmres(A,b,re4,tol,maxit,L,U);
re5 = 5;
[x5,fl5,rr5,it5,rv5] = gmres(A,b,re5,tol,maxit,L,U);
```

`fl3`, `fl4`, and `fl5` are all 0 because in each case restarted `gmres` drives the relative residual to less than the prescribed tolerance of `1e-12`. The following plots show the convergence histories of each restarted `gmres` method. `gmres(3)` converges at outer iteration 5, inner iteration 3 (`it3 = [5, 3]`) which would be the same as outer iteration 6, inner iteration 0, hence the marking of 6 on the final tick mark.


gmres(3)
Relative Residual
Iteration Number
10^0
10^-2
10^-4
10^-6
10^-8
10^-10
10^-12
10^-14
1
2
3
4
5
6


gmres(4)
Relative Residual
Iteration Number
10^0
10^-2
10^-4
10^-6
10^-8
10^-10
10^-12
10^-14
1
2
3



References

Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

Saad, Youcef and Martin H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, July 1986, Vol. 7, No. 3, pp. 856-869.

See Also

`bicg` | `bicgstab` | `cgs` | `lsqr` | `ilu` | `luinc` | `minres` | `pcg` | `qmr` | `symmlq` | `function_handle` | `mldivide`

gobjects

Purpose

Create array of graphics handles

Syntax

```
H = gobjects(n)
H = gobjects(s1,...,sn)
H = gobjects(v)

H = gobjects
H = gobjects(0)
```

Description

`H = gobjects(n)` returns an `n`-by-`n` array of graphics handles. Use the `gobjects` function instead of the `ones` or `zeros` functions to preallocate an array to store graphics handles.

`H = gobjects(s1,...,sn)` returns an `s1`-by-...-by-`sn` array of graphics handles, where the list of integers `s1,...,sn` defines the dimensions of the array. For example, `gobjects(2,3)` returns a 2-by-3 array.

`H = gobjects(v)` returns an array of graphics handles where the elements of the row vector, `v`, define the dimensions of the array. For example, `gobjects([2,3,4])` returns a 2-by-3-by-4 array.

`H = gobjects` returns a 1-by-1 graphics handle array.

`H = gobjects(0)` returns empty.

Input Arguments

n - Size of square matrix

integer value

Size of the square matrix, specified as an integer value. Negative integers are treated as 0. The square matrix has dimensions `n`-by-`n`.

Data Types

`single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

s1,...,sn - Size of each array dimension

two or more integer values

Size of each array dimension, specified as a list of two or more integer values. Negative integers are treated as 0.

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

v - Size of each array dimension

row vector of integer values

Size of each array dimension, specified as a row vector of integer values. Negative integers are treated as 0.

Example: [2,4,6,7]

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Examples

Specifying Array Dimensions

Preallocate a 4-by-1 array to store graphics handles.

```
H = gobjects(4,1);
```

You can assign axes handles to the array elements.

```
for k = 1:4
    H(k) = subplot(2,2,k);
end
```

Specifying Array Dimensions with Size of Existing Array

Create an array to store graphics handles using the size of an existing array.

Define `A` as a 3-by-4 array.

```
A = [1,2,3,2; 4,5,6,6; 7,8,9,7];
```

Create an array of graphics handles using the size of A.

```
v = size(A);
H = gobjects(v);
```

The dimensions of the graphics handle array are the same as the dimensions of A.

```
isequal(size(H),size(A))
```

```
ans =

     1
```

Returning Empty Handle Array

Use the `gobjects` function to return an empty array.

```
H = gobjects(0)
```

```
H =

     []
```

Purpose

Plot nodes and links representing adjacency matrix

Syntax

```
gplot(A,Coordinates)
gplot(A,Coordinates,LineSpec)
```

Description

The `gplot` function graphs a set of coordinates using an adjacency matrix.

`gplot(A,Coordinates)` plots a graph of the nodes defined in `Coordinates` according to the *n*-by-*n* adjacency matrix `A`, where *n* is the number of nodes. `Coordinates` is an *n*-by-2 matrix, where *n* is the number of nodes and each coordinate pair represents one node.

`gplot(A,Coordinates,`*`LineSpec`*`)` plots the nodes using the line type, marker symbol, and color specified by *`LineSpec`*.

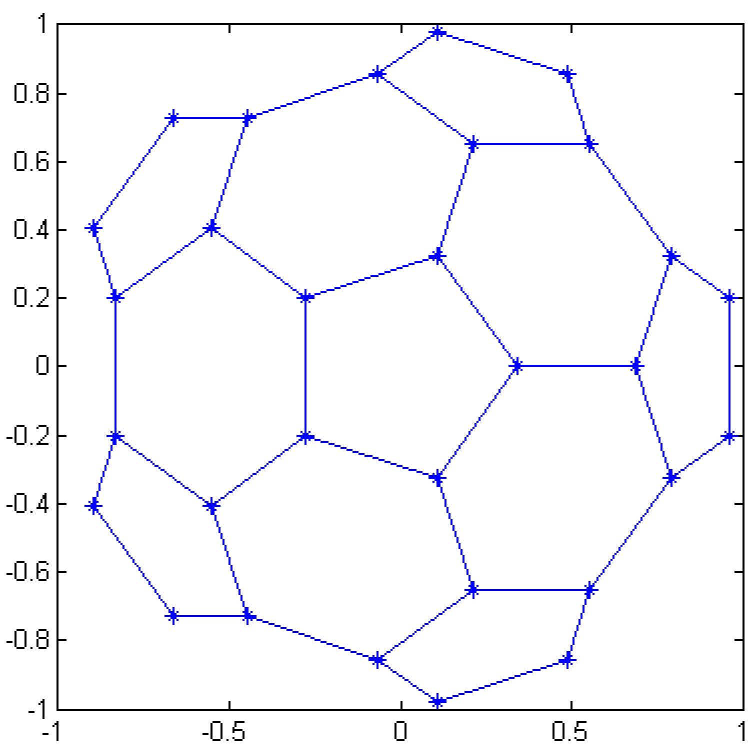
Tips

For two-dimensional data, `Coordinates(i,:) = [x(i) y(i)]` denotes node `i`, and `Coordinates(j,:) = [x(j)y(j)]` denotes node `j`. If node `i` and node `j` are connected, `A(i,j)` or `A(j,i)` is nonzero; otherwise, `A(i,j)` and `A(j,i)` are zero.

Examples

Plot half of a “Bucky ball” carbon molecule, placing asterisks at each node:

```
k = 1:30;
[B,XY] = bucky;
gplot(B(k,k),XY(k,:),'-*')
axis square
```



See Also

`LineSpec` | `sparse` | `spy`

Purpose

Extract MATLAB code from file published to HTML

Syntax

```
grabcode('name.html')
grabcode('urlname')
codeString = grabcode('name.html')
```

Description

`grabcode('name.html')` copies MATLAB code from the file `name.html` and pastes it into an untitled document in the Editor. Use `grabcode` to get MATLAB code from published files when the source code is not readily available. The file `name.html` was created by publishing `name.m`, a MATLAB code file containing cells. The MATLAB code from `name.m` is included at the end of `name.html` as HTML comments.

`grabcode('urlname')` copies MATLAB code from the `urlname` location and pastes it into an untitled document in the Editor.

`codeString = grabcode('name.html')` gets MATLAB code from the file `name.html` and assigns it the variable `codeString`.

Examples

This example illustrates how to use `grabcode` to get MATLAB code from an existing HTML file:

```
% Copy sine_wave_f.html and the images it includes
% from the MATLAB examples directory to your
% current folder:

copyfile(fullfile(matlabroot,'help','techdoc','matlab_env',...
 'examples','sine_wave_f.html'), 'my_sine_wave.html')

copyfile(fullfile(matlabroot,'help','techdoc','matlab_env',...
 'examples','sine_wave_f_01.png'), 'sine_wave_f_01.png')

copyfile(fullfile(matlabroot,'help','techdoc','matlab_env',...
 'examples','sine_wave_f_02.png'), 'sine_wave_f_02.png')

% If you want to view the file, double-click my_sine_wave.html
5 in your current folder.
```

```
% Extract the MATLAB code from sine_wave_f.html:
code = grabcode('my_sine_wave.html')
```

MATLAB returns:

```
code =

%% Plot Sine Wave
% Calculate and plot a sine wave.

%% Calculate and Plot Sine Wave
% Calculate and plot |y = sin(x)|.

function sine_wave_f(x)

y = sin(x);
plot(x,y)

%% Modify Plot Properties

title('Sine Wave', 'FontWeight','bold')
xlabel('x')
ylabel('sin(x)')
set(gca, 'Color', 'w')
set(gcf, 'MenuBar', 'none')

publish
```

Purpose

Numerical gradient

Syntax

```
FX = gradient(F)
[FX,FY] = gradient(F)
[FX,FY,FZ,...] = gradient(F)
[...] = gradient(F,h)
[...] = gradient(F,h1,h2,...)
```

Definitions

The *gradient* of a function of two variables, $F(x,y)$, is defined as

$$\nabla F = \frac{\partial F}{\partial x}\hat{i} + \frac{\partial F}{\partial y}\hat{j}$$

and can be thought of as a collection of vectors pointing in the direction of increasing values of F. In MATLAB software, numerical gradients (differences) can be computed for functions with any number of variables. For a function of N variables, $F(x,y,z, ...)$,

$$\nabla F = \frac{\partial F}{\partial x}\hat{i} + \frac{\partial F}{\partial y}\hat{j} + \frac{\partial F}{\partial z}\hat{k} + \ldots$$

Description

`FX = gradient(F)` where `F` is a vector returns the one-dimensional numerical gradient of `F`. `FX` corresponds to $\partial F/\partial x$, the differences in x (horizontal) direction.

`[FX,FY] = gradient(F)` where `F` is a matrix returns the x and y components of the two-dimensional numerical gradient. `FX` corresponds to $\partial F/\partial x$, the differences in x (horizontal) direction. `FY` corresponds to $\partial F/\partial y$, the differences in the y (vertical) direction. The spacing between points in each direction is assumed to be one.

`[FX,FY,FZ,...]  = gradient(F)` where `F` has `N dimensions` returns the `N` components of the gradient of `F`. There are two ways to control the spacing between values in `F`:

- A single spacing value, `h`, specifies the spacing between points in every direction.

- N spacing values (h1,h2,...) specifies the spacing for each dimension of F. Scalar spacing parameters specify a constant spacing for each dimension. Vector parameters specify the coordinates of the values along corresponding dimensions of F. In this case, the length of the vector must match the size of the corresponding dimension.

Note The first output FX is always the gradient along the 2nd dimension of F, going across columns. The second output FY is always the gradient along the 1st dimension of F, going across rows. For the third output FZ and the outputs that follow, the Nth output is the gradient along the Nth dimension of F.

[...] = gradient(F,h) where h is a scalar uses h as the spacing between points in each direction.

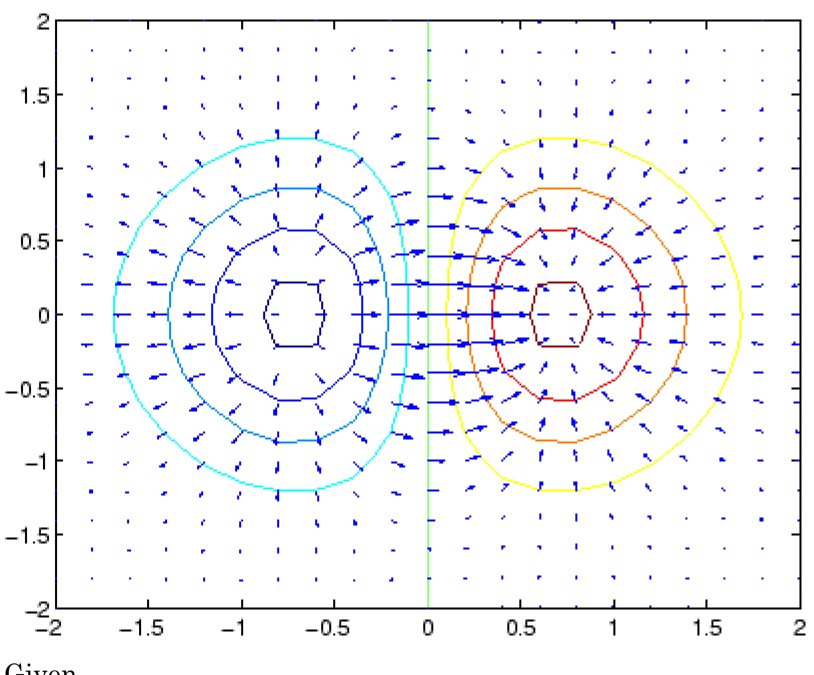
[...] = gradient(F,h1,h2,...) with N spacing parameters specifies the spacing for each dimension of F.

Examples

The statements

```
v = -2:0.2:2;
[x,y] = meshgrid(v);
z = x .* exp(-x.^2 - y.^2);
[px,py] = gradient(z,.2,.2);
contour(v,v,z), hold on, quiver(v,v,px,py), hold off
```

produce



Given,

```
F(:,:,1) = magic(3); F(:,:,2) = pascal(3);
gradient(F)
```

takes `dx = dy = dz = 1`.

```
[PX,PY,PZ] = gradient(F,0.2,0.1,0.2)
```

takes `dx = 0.2`, `dy = 0.1`, and `dz = 0.2`.

See Also

`del2` | `diff`

graymon

Purpose

Set default figure properties for grayscale monitors

Syntax

```
graymon
```

Description

`graymon` sets defaults for graphics properties to produce more legible displays for grayscale monitors.

See Also

`axes` | `figure`

Purpose

Grid lines for 2-D and 3-D plots

Syntax

```
grid on
grid off
grid minor
grid
grid(axes_handle,...)
grid minor
```

Description

`grid on` adds major grid lines to the current axes.

`grid off` removes major and minor grid lines from the current axes.

`grid minor` toggles the visibility of minor grid lines of the current axes.

`grid` toggles the major grid visibility state.

`grid(axes_handle,...)` uses the axes specified by `axes_handle` instead of the current axes.

Algorithms

`grid` sets the `XGrid`, `YGrid`, and `ZGrid` properties of the axes.

`grid minor` sets the `XMinorGrid`, `YMinorGrid`, and `ZMinorGrid` properties of the axes. Reissuing `grid minor` toggles minor grid visibility.

You can set the grid lines for just one axis using the `set` command and the individual property. For example,

```
set(axes_handle,'XGrid','on')
```

turns on only *x*-axis grid lines.

You can set grid line width with the axes `LineWidth` property.

By default, the number of grid lines changes when you resize a figure. To prevent this and keep grids the same at any size, set the `XTickMode` or `YTickMode` axes properties to `'manual'`:

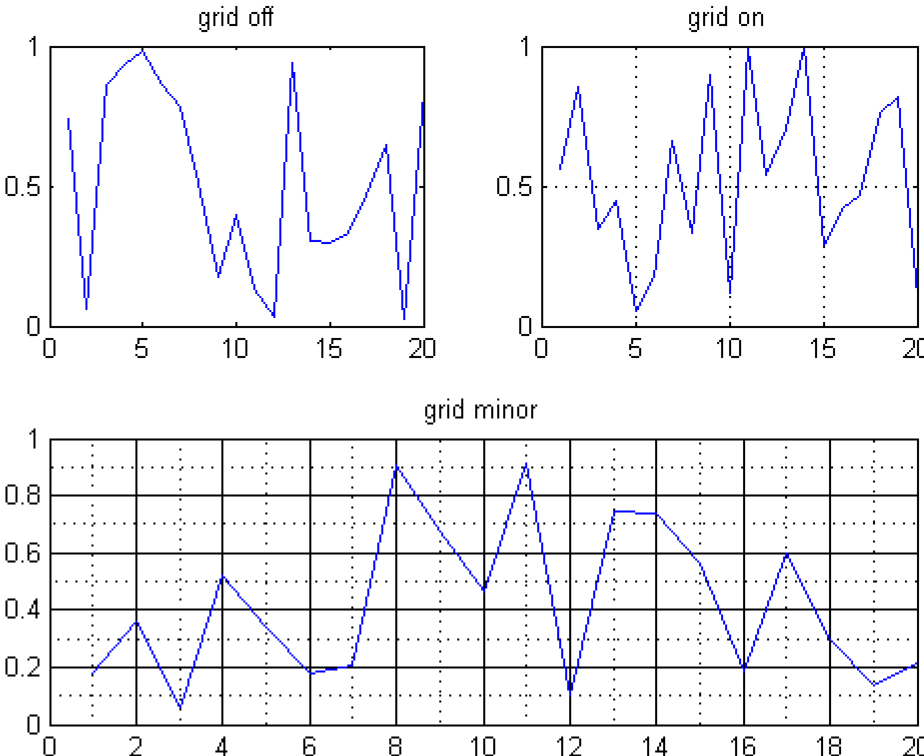
```
set(axes_handle,'XTickMode','manual')
set(axes_handle,'YTickMode','manual')
```

To customize the locations of ticks along an axis, set the axes XTick, YTick, and ZTick properties. For minor ticks, use the XMinorTick, YMinorTick, and ZTick properties.

Examples

Create multiple plots within a single figure and manipulate their grid visibility using the grid function:

```
figure
subplot(2,2,1)
plot(rand(1,20))
title('grid off')
subplot(2,2,2)
plot(rand(1,20))
grid on
title('grid on')
subplot(2,2,[3 4])
plot(rand(1,20))
grid on
set(gca, 'GridLineStyle', '-');
grid(gca,'minor')
title('grid minor')
```



You can also use the following code to control grids on a per-axis basis:

```
set(axh,'XGrid','on','YGrid','on','ZGrid','on')
```

Alternatives

To control the presence and appearance of grid lines on a graph, use the Property Editor, one of the plotting tools. For details, see The Property Editor.

See Also

box | axes | set

griddata

Purpose

Interpolate scattered data

Note Qhull-specific options are no longer supported. Remove the `OPTIONS` argument from all instances in your code that pass it to `griddata`.

In a future release, the following syntaxes will be removed:

```
[Xq,Yq,Vq] = griddata(x,y,v,xq,yq)
[Xq,Yq,Vq] = griddata(x,y,v,xq,yq, method)
```

In addition, `griddata` will not accept any input vectors of mixed orientation in a future release. To specify a grid of query points, construct a full grid with `ndgrid` or `meshgrid` before calling `griddata`.

Syntax

```
vq = griddata(x,y,v,xq,yq)
vq = griddata(x,y,z,v,xq,yq,zq)
vq = griddata(..., method)
```

Description

`vq = griddata(x,y,v,xq,yq)` fits a surface of the form $v = f(x,y)$ to the scattered data in the vectors `(x,y,v)`. The `griddata` function interpolates the surface at the query points specified by `(xq,yq)` and returns the interpolated values, `vq`. The surface always passes through the data points defined by `x` and `y`.

`vq = griddata(x,y,z,v,xq,yq,zq)` fits a hypersurface of the form $v = f(x,y,z)$.

`vq = griddata(..., method)` uses a specified interpolation method to compute `vq`.

Input Arguments

x

Vector specifying the *x*- coordinates of the sample points.

y

Vector specifying the *y*- coordinates of the sample points.

z

Vector specifying the *z*- coordinates of the sample points.

v

Vector of sample values that correspond to the sample coordinates `x`, `y` (and `z` for 3-D interpolation).

xq

Vector or array that specifies *x*- coordinates of the query points to be evaluated. `xq` must be the same size as `yq` (and `zq` for 3-D interpolation).

- Specify an array if you want to pass a grid of query points. Use `ndgrid` or `meshgrid` to construct the array.
- Specify a vector if you want to pass a collection of scattered points.

yq

Vector or array that specifies *y*- coordinates of the query points to be evaluated. `yq` must be the same size as `xq` (and `zq` for 3-D interpolation).

- Specify an array if you want to pass a grid of query points. Use `ndgrid` or `meshgrid` to construct the array.
- Specify a vector if you want to pass a collection of scattered points.

zq

Vector or array that specifies *z*- coordinates of the query points to be evaluated. `zq` must be the same size as `xq` and `yq`.

- Specify an array if you want to pass a grid of query points. Use `ndgrid` or `meshgrid` to construct the array.
- Specify a vector if you want to pass a collection of scattered points.

method

Keyword that specifies the interpolation method. Use one of the following:

`'linear'`	Linear interpolation (default)
`'cubic'`	Cubic interpolation
`'natural'`	Natural neighbor interpolation
`'nearest'`	Nearest neighbor interpolation
`'v4'`	MATLAB 4 `griddata` method

Output Arguments

vq

The interpolated values at the query points.

- For 2-D interpolation, where `xq` and `yq` specify an `m`-by-`n` grid of query points, `vq` is an `m`-by-`n` array.
- For 3-D interpolation, where `xq`, `yq`, and `zq` specify an `m`-by-`n`-by-`p` grid of query points, `vq` is an `m`-by-`n`-by-`p` array.
- If `xq`, `yq`, (and `zq` for 3-D interpolation) are vectors that specify scattered points, `vq` is a vector of the same length.

Examples

Interpolate Scattered Data Over a Uniform Grid

Sample a function at 200 random points between `-2.5` and `2.5`.

```
xy = -2.5 + 5*gallery('uniformdata',[200 2],0);
x = xy(:,1); y = xy(:,2);
v = x.*exp(-x.^2-y.^2);
```

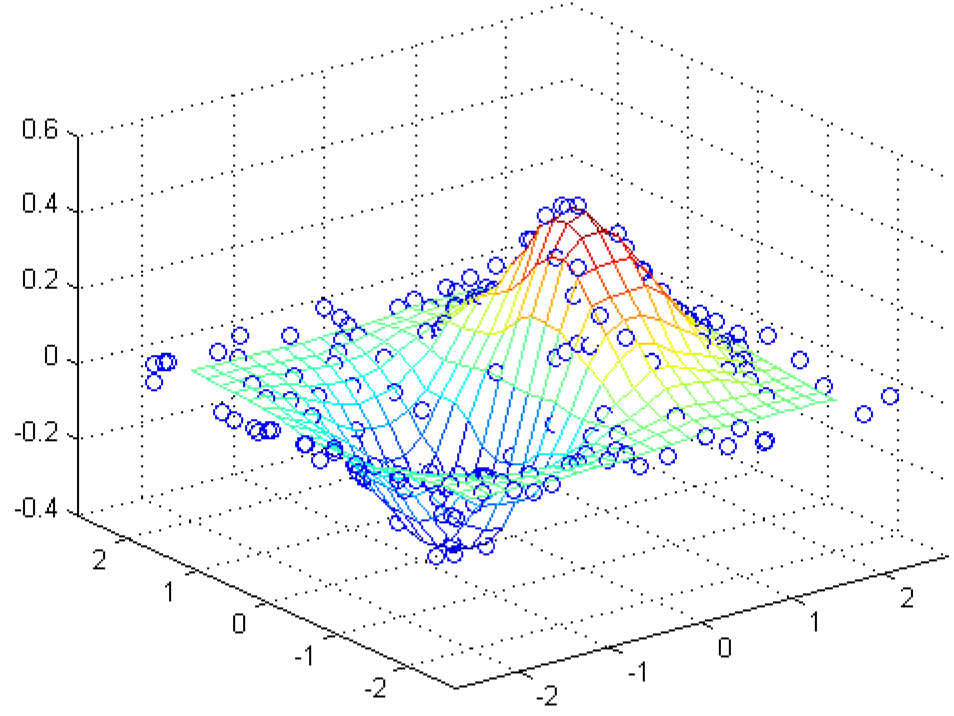
`x`, `y`, and `v` are vectors containing scattered (nonuniform) sample points and data.

Define a regular grid and interpolate the scattered data over the grid.

```
[xq,yq] = meshgrid(-2:.2:2, -2:.2:2);
vq = griddata(x,y,v,xq,yq);
```

Plot the gridded data as a mesh and the scattered data as dots.

```
figure
mesh(xq,yq,vq);
hold on
plot3(x,y,v,'o');
h = gca;
set(h,'XLim',[-2.7 2.7]);
set(h,'YLim',[-2.7 2.7]);
```



Interpolate 3-D Data Set Over a Grid in the x-y Plane

Sample a function at 5000 random points between -1 and 1.

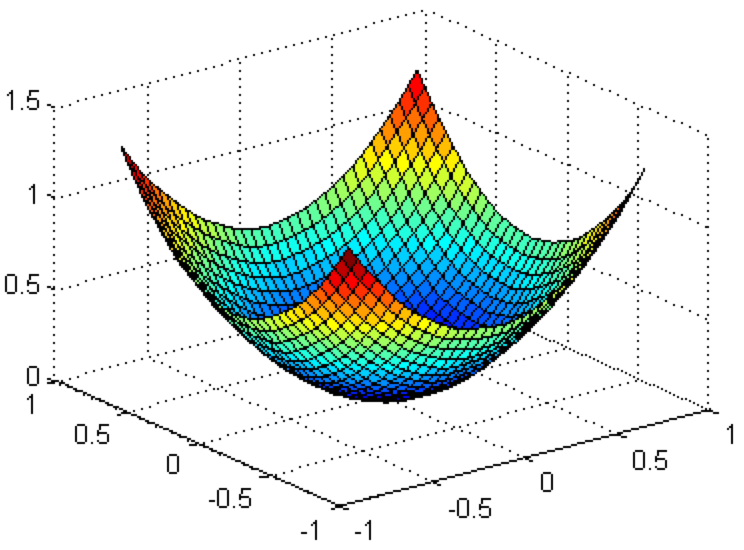
```
rng(0,'twister')
x = 2*rand(5000,1)-1;
y = 2*rand(5000,1)-1;
z = 2*rand(5000,1)-1;
v = x.^2 + y.^2 + z.^2;
```

x, y, and z are now vectors containing nonuniformly sampled data. Define a regular grid with points in the range [-0.8, 0.8].

```
d = -0.8:0.05:0.8;
[xq,yq,zq] = meshgrid(d,d,0);
```

Interpolate the scattered data over a rectangular region at z=0. Then, plot the results.

```
vq = griddata(x,y,z,v,xq,yq,zq);
surf(xq,yq,vq);
set(gca,'XTick',[-1 -0.5 0 0.5 1]);
set(gca,'YTick',[-1 -0.5 0 0.5 1]);
```



See Also

`scatteredInterpolant` | `delaunay` | `griddatan` | `interpn` | `meshgrid` | `ndgrid`

griddatan

Purpose

Data gridding and hypersurface fitting (dimension ≥ 2)

Syntax

```
yi = griddatan(x,y,xi)
yi = griddatan(x,y,xi,method)
yi = griddatan(x,y,xi,method,options)
```

Description

`yi = griddatan(x,y,xi)` fits a hyper-surface of the form *y = f(x)* to the data in the (usually) nonuniformly-spaced vectors (`x`, `y`). `griddatan` interpolates this hyper-surface at the points specified by `xi` to produce `yi`. `xi` can be nonuniform.

`X` is of dimension `m`-by-`n`, representing `m` points in n-dimensional space. `y` is of dimension `m`-by-`1`, representing `m` values of the hyper-surface *f*(`X`). `xi` is a vector of size `p`-by-`n`, representing `p` points in the n-dimensional space whose surface value is to be fitted. `yi` is a vector of length `p` approximating the values *f*(`xi`). The hypersurface always goes through the data points (`X`,`y`). `xi` is usually a uniform grid (as produced by `meshgrid`).

`yi = griddatan(x,y,xi,method)` defines the type of surface fit to the data, where `'method'` is one of:

'linear'	Triangulation-based linear interpolation (default)
'nearest'	Nearest neighbor interpolation

All the methods are based on a Delaunay triangulation of the data.

If `method` is `[]`, the default `'linear'` method is used.

`yi = griddatan(x,y,xi,method,options)` specifies a cell array of strings `options` to be used in Qhull via `delaunayn`.

If `options` is `[]`, the default options are used. If `options` is `{''}`, no options are used, not even the default.

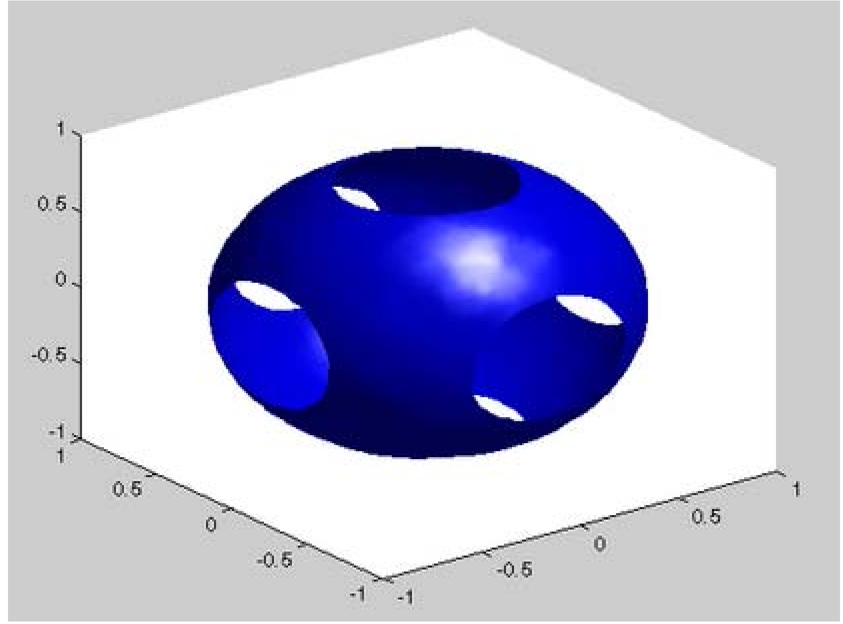
Examples

```
X=2*gallery('uniformdata',[5000 3],0)-1;;
Y = sum(X.^2,2);
d = -0.8:0.05:0.8;
```

```
[y0,x0,z0] = ndgrid(d,d,d);
XI = [x0(:) y0(:) z0(:)];
YI = griddatan(X,Y,XI);
```

Since it is difficult to visualize 4-D data sets, use isosurface at 0.8:

```
YI = reshape(YI, size(x0));
p = patch(isosurface(x0,y0,z0,YI,0.8));
isonormals(x0,y0,z0,YI,p);
set(p,'FaceColor','blue','EdgeColor','none');
view(3), axis equal, axis off, camlight, lighting phong
```



See Also

delaunayn | griddata | meshgrid

griddedInterpolant

Purpose

Gridded data interpolation

Note The behavior of `griddedInterpolant` has changed. All interpolation methods now support extrapolation by default. Set `F.ExtrapolationMethod` to `'none'` to preserve the pre-R2013a behavior when `F.Method` is `'linear'`, `'cubic'` or `'nearest'`. Before R2013a, evaluation returned `NaN` values at query points outside the domain when `F.Method` was set to `'linear'`, `'cubic'` or `'nearest'`.

Description

Use `griddedInterpolant` to perform interpolation on a 1-D, 2-D, 3-D, or N-D "Gridded Data" on page 1-2062 set. For example, you can pass a set of `(x,y)` points and values, `v`, to `griddedInterpolant`, and it returns a surface of the form $v = F(x, y)$. This surface always passes through the sample values at the point locations. You can evaluate this surface at any query point, `(xq,yq)`, to produce an interpolated value, `vq`.

Use `griddedInterpolant` to create the "Interpolant" on page 1-2062, `F`. Then you can evaluate `F` at specific points using any of the following syntaxes:

- `Vq = F(Xq)` evaluates `F` at a set of query points in matrix `Xq`. The points in `Xq` are scattered, and each row of `Xq` contains the coordinates of a query point.
- `Vq = F(xq1,xq2,...,xqn)` specifies the query locations, `xq1,xq2,...,xqn`, as column vectors of length `m` representing `m` points scattered in `n`-dimensional space.
- `Vq = F(Xq1,Xq2,...,Xqn)` specifies the query locations as `n` `n`-dimensional arrays, `Xq1,Xq2,...,Xqn`, of equal size which define a "Full Grid" on page 1-2062 of points.
- `Vq = F({xgq1,xgq2,...,xgqn})` specifies the query locations as "Grid Vectors" on page 1-2062 . Use this syntax to conserve memory when you want to query a large grid of points.

Construction

`F = griddedInterpolant(x,v)` creates a 1-D interpolant from a vector of sample points, `x`, and corresponding values, `v`.

`F = griddedInterpolant(X1,X2,...,Xn,V)` creates a 2-D, 3-D, or N-D interpolant using a "Full Grid" on page 1-2062 of sample points passed as a set of `n`-dimensional arrays, `X1,X2,...,Xn`. The V array contains the sample values associated with the point locations in `X1,X2,...,Xn`. Each of the arrays, `X1,X2,...,Xn` must be the same size as `V`.

`F = griddedInterpolant(V)` uses the default grid to create the interpolant. When you use this syntax, `griddedInterpolant` defines the grid as set of points whose spacing is `1` and range is [`1`, `size(V,i)`] in the `i`th dimension. Use this syntax to when you want to conserve memory and are not concerned about the relative distances between points.

`F = griddedInterpolant({xg1,xg2,...,xgn},V)` specifies `n` "Grid Vectors" on page 1-2062 to describe an `n`-dimensional grid of sample points. Use this syntax when you want to use a specific grid and also conserve memory.

`F = griddedInterpolant( ___ ,Method)` specifies any of five strings that describe an interpolation method: `'nearest'`, `'linear'`, `'spline'`, `'pchip'`, or `'cubic'`. You can specify `Method` as the last input argument in any of the first four syntaxes.

`F = griddedInterpolant( ___ ,Method,ExtrapolationMethod)` specifies both the interpolation and extrapolation methods as strings. `griddedInterpolant` uses `ExtrapolationMethod` to estimate the value when your query points fall outside the domain of your sample points. Specify `Method` and `ExtrapolationMethod` together as the last two input arguments in any of the first four syntaxes.

Input Arguments

x

Sample points vector, specified as a vector of input coordinates the same size as `v`.

v

Sample values vector, specified as a vector of input values the same size as x.

X1,X2,...,Xn

Sample points in "Full Grid" on page 1-2062 form, specified as a set of n-dimensional arrays. You can create the arrays, X1,X2,...,Xn, using the ndgrid function. These arrays are all the same size, and each one is the same size as V.

{xg1,xg2,...,xgn}

Sample points in grid vector form, specified as a cell array of grid vectors. These vectors must specify a grid that is the same size as V. In other words, size(V) = [length(xg1) length(xg2),...,length(xgn)]. Use this form as an alternative to the full grid to save memory when your grid is very large.

V

Sample values, specified as an array. The elements of V are the values that correspond to the sample points. The size of V must be the size of the full grid of sample points.

- If you specify the sample points as a full grid consisting of N-D arrays, then V must be the same size as any one of: X1,X2,...,Xn.
- If you specify the sample points as grid vectors, then size(V) = [length(xg1) length(xg2) ... length(xgn)].

Method

Interpolation method, specified as a string from the table below.

Method String	Description
'linear' (default)	Linear interpolation of the values at neighboring grid points in each respective dimension.
'nearest'	Nearest neighbor interpolation.

Method String	Description
'spline'	Spline interpolation.
'pchip'	Shape-preserving piecewise cubic interpolation (1-D only).
'cubic'	Cubic interpolation of the values at neighboring grid points in each respective dimension. This method supports only uniformly spaced data.

ExtrapolationMethod

Extrapolation method, specified as any of the `Method` choices: `'linear'`, `'nearest'`, `'spline'`, `'pchip'`, or `'cubic'`. In addition, you can specify `'none'` if you want queries outside the domain of your grid return `NaN` values.

If you omit `ExtrapolationMethod`, the default value is the string you specify for `Method`. If you omit both the `Method` and `ExtrapolationMethod` arguments, both default to `'linear'`.

Properties

GridVectors

Cell array containing grid vectors, `{xg1,xg2,...,xgn}`. These vectors specify the grid points (locations) for the values in `F.Values`.

Values

Array of values associated with the grid points in `F.GridVectors`.

Method

A string specifying the name of a method used to interpolate the data. `Method` is one of five strings: `'nearest'`, `'linear'`, `'spline'`, `'pchip'`, or `'cubic'`. The default value is `'linear'`.

ExtrapolationMethod

A string specifying the name of a method used to extrapolate the data. `ExtrapolationMethod` is one of six strings: `'nearest'`,

`'linear'`, `'spline'`, `'pchip'`, `'cubic'`, or `'none'`. A value of `'none'` indicates that extrapolation is disabled. The default value is the value of `F.Method`.

Definitions

Interpolant

Interpolating function that you can evaluate at query locations.

Gridded Data

A set of points that are axis-aligned and ordered.

Scattered Data

A set of points that have no structure among their relative locations.

Full Grid

A grid represented as a set of arrays. For example, you can create a full grid using `ndgrid`.

Grid Vectors

A set of vectors that serve as a compact representation of a grid in `ndgrid` format. For example, `[X,Y] = ndgrid(xg,yg)` returns a full grid in the matrices `X` and `Y`. You can represent the same grid using the grid vectors, `xg` and `yg`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Indexing

Index-based editing of the properties of `F` are not supported. Instead, wholly replace the `GridVectors` or `Values` arrays as necessary. See "Interpolation with the griddedInterpolant Class" in the MATLAB Mathematics documentation for more information.

Examples

2-D Interpolation Over Finer Grid

Interpolate coarsely sampled data using a full grid with spacing of `0.5`.

Define the sample points as a full grid with range [1, 10] in both dimensions.

```
[X,Y] = ndgrid(1:10,1:10);
```

Sample $f(x,y) = x^2+y^2$ at the grid points.

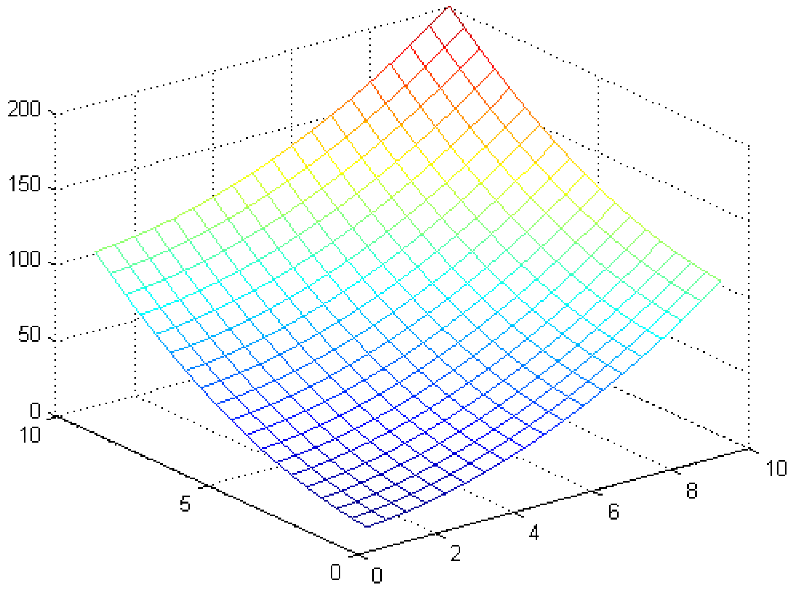
```
V = X.^2 + Y.^2;
```

Create the interpolant, specifying cubic interpolation.

```
F = griddedInterpolant(X,Y,V,'cubic');
```

Define a full grid of query points with `0.5` spacing and evaluate the interpolant at those points. Then plot the result.

```
[Xq,Yq] = ndgrid(1:0.5:10,1:0.5:10);
Vq = F(Xq,Yq);
mesh(Xq,Yq,Vq);
```



1-D Extrapolation

Compare results of querying the interpolant outside the domain of `F` using the `'pchip'` and `'nearest'` extrapolation methods.

Create the interpolant and specify `'pchip'` as the interpolation method.

```
x = [1 2 3 4 5];
v = [12 16 31 10 6];
F = griddedInterpolant(x,v,'pchip')
```

```
F =

  griddedInterpolant with properties:

            GridVectors: {[1 2 3 4 5]}
                 Values: [12 16 31 10 6]
                 Method: 'pchip'
```

```
    ExtrapolationMethod: 'pchip'
```

Query the interpolant, and include points outside the domain of `F`.

```
xq = 0:0.1:6;
vq = F(xq);
figure
plot(x,v,'o',xq,vq,'-b');
legend ('v','vq')
```

Query the interpolant at the same points again, using the nearest neighbor extrapolation method.

```
F.ExtrapolationMethod = 'nearest';
figure
vq = F(xq);
plot(x,v,'o',xq,vq,'-b');
legend ('v','vq')
```

See Also

`scatteredInterpolant` | `interp1` | `interp2` | `interp3` | `interpn` | `ndgrid` | `meshgrid`

How To

- Class Attributes
- Property Attributes
- "Interpolating Gridded Data"

gsvd

Purpose

Generalized singular value decomposition

Syntax

```
[U,V,X,C,S] = gsvd(A,B)
sigma = gsvd(A,B)
```

Description

`[U,V,X,C,S] = gsvd(A,B)` returns unitary matrices `U` and `V`, a (usually) square matrix `X`, and nonnegative diagonal matrices `C` and `S` so that

```
A = U*C*X'
B = V*S*X'
C'*C + S'*S = I
```

`A` and `B` must have the same number of columns, but may have different numbers of rows. If `A` is `m`-by-`p` and `B` is `n`-by-`p`, then `U` is `m`-by-`m`, `V` is `n`-by-`n` and `X` is `p`-by-`q` where `q = min(m+n,p)`.

`sigma = gsvd(A,B)` returns the vector of generalized singular values, `sqrt(diag(C'*C)./diag(S'*S))`.

The nonzero elements of `S` are always on its main diagonal. If `m >= p` the nonzero elements of `C` are also on its main diagonal. But if `m < p`, the nonzero diagonal of `C` is `diag(C,p-m)`. This allows the diagonal elements to be ordered so that the generalized singular values are nondecreasing.

`gsvd(A,B,0)`, with three input arguments and either `m` or `n >= p`, produces the "economy-sized"decomposition where the resulting `U` and `V` have at most `p` columns, and `C` and `S` have at most `p` rows. The generalized singular values are `diag(C)./diag(S)`.

When `B` is square and nonsingular, the generalized singular values, `gsvd(A,B)`, are equal to the ordinary singular values, `svd(A/B)`, but they are sorted in the opposite order. Their reciprocals are `gsvd(B,A)`.

In this formulation of the `gsvd`, no assumptions are made about the individual ranks of `A` or `B`. The matrix `X` has full rank if and only if the matrix `[A;B]` has full rank. In fact, `svd(X)` and `cond(X)` are equal to `svd([A;B])` and `cond([A;B])`. Other formulations, eg. G. Golub and

C. Van Loan [1], require that `null(A)` and `null(B)` do not overlap and replace `X` by `inv(X)` or `inv(X')`.

Note, however, that when `null(A)` and `null(B)` do overlap, the nonzero elements of `C` and `S` are not uniquely determined.

Examples

Example 1

The matrices have at least as many rows as columns.

```
A = reshape(1:15,5,3)
B = magic(3)
A =
          1     6    11
          2     7    12
          3     8    13
          4     9    14
          5    10    15
B =
          8     1     6
          3     5     7
          4     9     2
```

The statement

```
[U,V,X,C,S] = gsvd(A,B)
```

produces a 5-by-5 orthogonal `U`, a 3-by-3 orthogonal `V`, a 3-by-3 nonsingular `X`,

```
X =
        2.8284   -9.3761   -6.9346
       -5.6569   -8.3071  -18.3301
        2.8284   -7.2381  -29.7256
```

and

```
C =
        0.0000         0         0
```

```
              0    0.3155         0
              0         0    0.9807
              0         0         0
              0         0         0
S =
         1.0000         0         0
              0    0.9489         0
              0         0    0.1957
```

Since `A` is rank deficient, the first diagonal element of `C` is zero.

The economy sized decomposition,

```
[U,V,X,C,S] = gsvd(A,B,0)
```

produces a 5-by-3 matrix `U` and a 3-by-3 matrix `C`.

```
U =
         0.5700   -0.6457   -0.4279
        -0.7455   -0.3296   -0.4375
        -0.1702   -0.0135   -0.4470
         0.2966    0.3026   -0.4566
         0.0490    0.6187   -0.4661

C =
         0.0000         0         0
              0    0.3155         0
              0         0    0.9807
```

The other three matrices, `V`, `X`, and `S` are the same as those obtained with the full decomposition.

The generalized singular values are the ratios of the diagonal elements of `C` and `S`.

```
sigma = gsvd(A,B)
sigma =
         0.0000
         0.3325
```

```
        5.0123
```

These values are a reordering of the ordinary singular values

```
svd(A/B)
ans =
        5.0123
        0.3325
        0.0000
```

Example 2

The matrices have at least as many columns as rows.

```
A = reshape(1:15,3,5)
B = magic(5)
A =

     1     4     7    10    13
         2     5     8    11    14
         3     6     9    12    15
B =
    17    24     1     8    15
        23     5     7    14    16
         4     6    13    20    22
        10    12    19    21     3
        11    18    25     2     9
```

The statement

```
[U,V,X,C,S] = gsvd(A,B)
```

produces a 3-by-3 orthogonal `U`, a 5-by-5 orthogonal `V`, a 5-by-5 nonsingular `X` and

```
C =
             0          0     0.0000          0          0
             0          0          0     0.0439          0
             0          0          0          0     0.7432
```

```
S =
       1.0000         0         0         0         0
            0    1.0000         0         0         0
            0         0    1.0000         0         0
            0         0         0    0.9990         0
            0         0         0         0    0.6690
```

In this situation, the nonzero diagonal of `C` is `diag(C,2)`. The generalized singular values include three zeros.

```
sigma = gsvd(A,B)
sigma =
            0
            0
       0.0000
       0.0439
       1.1109
```

Reversing the roles of `A` and `B` reciprocates these values, producing two infinities.

```
gsvd(B,A)
ans =
      1.0e+016 *

       0.0000
       0.0000
       4.4126
          Inf
          Inf
```

Algorithms

The generalized singular value decomposition uses the C-S decomposition described in [1], as well as the built-in `svd` and `qr` functions. The C-S decomposition is implemented in a local function in the `gsvd` program file.

Diagnostics

The only warning or error message produced by `gsvd` itself occurs when the two input arguments do not have the same number of columns.

References

[1] Golub, Gene H. and Charles Van Loan, *Matrix Computations*, Third Edition, Johns Hopkins University Press, Baltimore, 1996

See Also

`qr` | `svd`

Purpose

Test for greater than

Syntax

```
A > B
gt(A, B)
```

Description

`A > B` compares each element of array `A` with the corresponding element of array `B`, and returns an array with elements set to logical 1 (`true`) where `A` is greater than `B`, or set to logical 0 (`false`) where `A` is less than or equal to `B`. Each input of the expression can be an array or a scalar value.

If both `A` and `B` are scalar (i.e., 1-by-1 matrices), then the MATLAB software returns a scalar value.

If both `A` and `B` are nonscalar arrays, then these arrays must have the same dimensions, and MATLAB returns an array of the same dimensions as `A` and `B`.

If one input is scalar and the other a nonscalar array, then the scalar input is treated as if it were an array having the same dimensions as the nonscalar input array. In other words, if input `A` is the number 100, and `B` is a 3-by-5 matrix, then `A` is treated as if it were a 3-by-5 matrix of elements, each set to 100. MATLAB returns an array of the same dimensions as the nonscalar input array.

`gt(A, B)` is called for the syntax `A>B` when either `A` or `B` is an object.

Examples

Create two 6-by-6 matrices, `A` and `B`, and locate those elements of `A` that are greater than the corresponding elements of `B`:

```
A = magic(6);
B = repmat(3*magic(3), 2, 2);

A > B
ans =
     1     0     0     1     1     1
     0     1     0     1     1     1
     1     0     0     1     0     1
     0     1     1     0     1     0
```

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

See Also

`lt` | `ge` | `le` | `ne` | `eq`

How To

- "Relational Operators"

gtext

Purpose

Mouse placement of text in 2-D view

Syntax

```
gtext('string')
gtext({'string1','string2','string3',...})
gtext({'string1';'string2';'string3';...})
gtext(...,'PropertyName',PropertyValue,...)
h = gtext(...)
```

Description

`gtext` displays a text string in the current figure window after you select a location with the mouse.

`gtext('`*`string`*`')` waits for you to press a mouse button or keyboard key while the pointer is within a figure window. Pressing a mouse button or any key places `'`*`string`*`'` on the plot at the selected location.

`gtext({'string1','string2','string3',...})` places all strings with one click, each on a separate line.

`gtext({'string1';'string2';'string3';...})` places one string per click, in the sequence specified.

`gtext(...,'PropertyName',PropertyValue,...)` sets the values of the specified text properties. For a list of properties, see `Text Properties`.

`h = gtext(...)` returns the handle to a text graphics object that is placed on the plot at the location you select.

Tips

As you move the pointer into a figure window, the pointer becomes crosshairs to indicate that `gtext` is waiting for you to select a location. `gtext` uses the functions `ginput` and `text`.

Examples

Place a label on the current plot:

```
gtext('Note this divergence!')
```

See Also

`ginput` | `text`

Purpose

Store or retrieve GUI data

Syntax

```
guidata(object_handle,data)
data = guidata(object_handle)
```

Description

`guidata(object_handle,data)` stores the variable `data` as GUI data. If `object_handle` is not a figure handle, then the object's parent figure is used. `data` can be any MATLAB variable, but is typically a structure, which enables you to add new fields as required.

`guidata` can manage only one variable at any time. Subsequent calls to `guidata(object_handle,data)` overwrite the previously created version of GUI `data`.

GUIDE Uses guidata

GUIDE uses `guidata` to store and maintain the `handles` structure. In a GUIDE GUI code file, do not overwrite the `handles` structure or your GUI will no longer work. If you need to store data other than handles for your GUI, you can add new fields to the `handles` structure and safely place your data there. See GUI Data in the MATLAB documentation.

`data = guidata(object_handle)` returns previously stored data, or an empty matrix if nothing is stored.

To change the data managed by `guidata`:

1. Get a copy of the data with the command `data = guidata(object_handle)`.
2. Make the desired changes to `data`.
3. Save the changed version of `data` with the command `guidata(object_handle,data)`.

guidata provides application developers with a convenient interface to a figure's application data:

- You do not need to create and maintain a hard-coded property name for the application data throughout your source code.
- You can access the data from within a local function callback routine using the component's handle (which is returned by gcbo), without needing to find the figure's handle.

If you are not using GUIDE, guidata is particularly useful in conjunction with guihandles, which creates a structure containing the handles of all the components in a GUI.

Examples

This example calls guidata to save a structure containing a GUI figure's application data from within the initialization section of the application code file. The first section shows how to do this within a GUI you create manually. The second section shows how the code differs when you use GUIDE to create a template code file. GUIDE provides a handles structure as an argument to all local function callbacks, so you do not need to call guidata to obtain it. You do, however, need to call guidata to save changes you make to the structure.

Using guidata in a Programmed GUI

Calling the guihandles function creates the structure into which your code places additional data. It contains all handles used by the figure at the time it is called, generating field names based on each object's Tag property.

```
% Create figure to use as GUI in your main function or a local function
figure_handle = figure('Toolbar','none');
% create structure of handles
myhandles = guihandles(figure_handle);
% Add some additional data as a new field called numberOfErrors
myhandles.numberOfErrors = 0;
% Save the structure
guidata(figure_handle,myhandles)
```

You can recall the data from within a local callback function, modify it, and then replace the structure in the figure:

```
function My_Callback()
% ...
% Get the structure using guidata in the local function
myhandles = guidata(gcbo);
% Modify the value of your counter
myhandles.numberOfErrors = myhandles.numberOfErrors + 1;
% Save the change you made to the structure
guidata(gcbo,myhandles)
```

Using guidata in a GUIDE GUI

If you use GUIDE, you do not need to call `guihandles` to create a structure, because GUIDE generates a `handles` structure that contains the GUI's handles. You can add your own data to it, for example from within the `OpeningFcn` template that GUIDE creates:

```
% --- Executes just before simple_gui_tab is made visible.
function my_GUIDE_GUI_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to simple_gui_tab (see VARARGIN)
% ...

% add some additional data as a new field called numberOfErrors
handles.numberOfErrors = 0;
% Save the change you made to the structure
guidata(hObject,handles)
```

Notice that you use the input argument `hObject` in place of `gcbo` to refer to the object whose callback is executing.

Suppose you needed to access the `numberOfErrors` field in a push button callback. Your callback code now looks something like this:

```
% --- Executes on button press in pushbutton1.
function my_GUIDE_GUI_pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% ...

% No need to call guidata to obtain a structure;
% it is provided by GUIDE via the handles argument
handles.numberOfErrors = handles.numberOfErrors + 1;
% save the changes to the structure
guidata(hObject,handles)
```

See Also

guide | guihandles | getappdata | setappdata

Purpose

Open GUI Layout Editor

Syntax

```
guide
guide('filename.fig')
guide('fullpath')
guide(HandleList)
```

Description

`guide` initiates the GUI design environment (GUIDE) tools that allow you to create or edit GUIs interactively.

`guide` opens the GUIDE Quick Start dialog where you can choose to open a previously created GUI or create a new one using one of the provided templates.

`guide('filename.fig')` opens the FIG-file named `filename.fig` for editing if it is on the MATLAB path.

`guide('fullpath')` opens the FIG-file at `fullpath` even if it is not on the MATLAB path.

`guide(HandleList)` opens the content of each of the figures in `HandleList` in a separate copy of the GUIDE design environment.

See Also

`inspect`

How To

- "Ways to Build MATLAB GUIs"
- "About the Simple GUIDE GUI"

guihandles

Purpose

Create structure of handles

Syntax

```
handles = guihandles(object_handle)
handles = guihandles
```

Description

`handles = guihandles(object_handle)` returns a structure containing the handles of the objects in a figure, using the value of their `Tag` properties as the fieldnames, with the following caveats:

- Objects are excluded if their `Tag` properties are empty, or are not legal variable names.
- If several objects have the same `Tag`, that field in the structure contains a vector of handles.
- Objects with hidden handles are included in the structure.

`handles = guihandles` returns a structure of handles for the current figure.

See Also

`guidata` | `guide` | `getappdata` | `setappdata`

Purpose

Uncompress GNU zip files

Syntax

```
gunzip(files)
gunzip(files,outputdir)
gunzip(url, ...)
filenames = gunzip(...)
```

Description

`gunzip(files)` uncompresses GNU zip files from the list of files specified in `files`. Directories recursively `gunzip` all of their content. The output files have the same name, excluding the extension `.gz`, and are written to the same directory as the input files.

`files` is a string or cell array of strings containing a list of files or directories. Individual files that are on the MATLAB path can be specified as partial path names. Otherwise, an individual file can be specified relative to the current directory or with an absolute path.

Folders must be specified relative to the current folder or with absolute paths. On UNIX systems, folders can also start with `~/` or `~`*`username`*`/`, which expands to the current user's home folder or the specified user's home folder, respectively. The wildcard character `*` can be used when specifying files or folders, except when relying on the MATLAB path to resolve a file name or partial path name.

`gunzip(files,outputdir)` writes the gunzipped file into the directory `outputdir`. If `outputdir` does not exist, MATLAB creates it.

`gunzip(url, ...)` extracts the GNU zip contents from an Internet universal resource locator (URL). The URL must include the protocol type (for example, `'http://'`). MATLAB downloads the URL to the temp directory, and then deletes it.

`filenames = gunzip(...)` gunzips the files and returns the relative path names of the gunzipped files in the string cell array `filenames`.

Examples

To `gunzip` all `.gz` files in the current directory, type:

```
gunzip('*.gz');
```

To `gunzip` Cleve Moler's "Numerical Computing with MATLAB" examples to the output directory `ncm`, type:

```
url ='http://www.mathworks.com/moler/ncm.tar.gz';
gunzip(url,'ncm')
untar('ncm/ncm.tar','ncm')
```

See Also

`gzip` | `tar` | `untar` | `unzip` | `zip`

Purpose

Compress files into GNU zip files

Syntax

```
gzip(files)
gzip(files,outputdir)
filenames = gzip(...)
```

Description

`gzip(files)` creates GNU zip files from the list of files specified in `files`. Directories recursively `gzip` all their contents. Each output gzipped file is written to the same directory as the input file and with the file extension `.gz`.

`files` is a string or cell array of strings containing a list of files or directories to `gzip`. Individual files that are on the MATLAB path can be specified as partial path names. Otherwise, an individual file can be specified relative to the current directory or with an absolute path.

Folders must be specified relative to the current folder or with absolute paths. On UNIX systems, folders can also start with `~/` or `~`*`username`*`/`, which expands to the current user's home folder or the specified user's home folder, respectively. The wildcard character `*` can be used when specifying files or folders, except when relying on the MATLAB path to resolve a file name or partial path name.

`gzip(files,outputdir)` writes the gzipped files into the directory `outputdir`. If `outputdir` does not exist, MATLAB creates it.

`filenames = gzip(...)` gzips the files and returns the relative path names of all gzipped files in the string cell array `filenames`.

Examples

To `gzip` all `.m` and `.mat` files in the current directory and store the results in the directory `archive`, type:

```
gzip({'*.m','*.mat'},'archive');
```

See Also

`gunzip` | `tar` | `untar` | `unzip` | `zip`

h5create

Purpose

Create HDF5 data set

Syntax

```
h5create(filename,datasetname,size,Name,Value)
```

Description

`h5create(filename,datasetname,size,Name,Value)` creates an HDF5 data set in the file specified by `filename`.

Input Arguments

filename

Text string specifying the name of an HDF5 file. If `filename` does not already exist, `h5create` creates it, with additional options specified by one or more `Name,Value` pair arguments.

datasetname

Text string specifying the name of the data set you want to create. If `datasetname` is a full path name, `h5create` creates all intermediate groups, if they don't already exist.

size

Array specifying the extents of the dataset. To specify an unlimited extent, set the corresponding element of `size` to `Inf`.

Name-Value Pair Arguments

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

Datatype

Any of the following MATLAB datatypes.

double	uint64	uint32	uint16	uint8
single	int64	int32	int16	int8

Default: `double`

ChunkSize

Defines chunking layout.

Default: Not chunked

Deflate

Defines gzip compression level (0-9).

Default: 0

FillValue

Defines the fill value for numeric data sets.

Fletcher32

Turns on the Fletcher32 checksum filter.

Default: `false`

Shuffle

Turns on the Shuffle filter.

Default: `false`

Examples

Create a fixed-size 100-by-200 data set.

```
h5create('myfile.h5','/myDataset1',[100 200]);
h5disp('myfile.h5');
```

Create a single-precision 1000-by-2000 data set with a chunk size of 50-by-80. Apply the highest level of compression.

```
h5create('myfile.h5','/myDataset2',[1000 2000], 'Datatype','single', ...
         'ChunkSize',[50 80],'Deflate',9);
h5disp('myfile.h5');
```

Create a two-dimensional data set that is unlimited along the second extent.

```
h5create('myfile.h5','/myDataset3',[200 Inf],'ChunkSize',[20 20]);
h5disp('myfile.h5');
```

See Also

h5read | h5write | h5info | h5disp

Tutorials

- “Exporting to Hierarchical Data Format (HDF5) Files”

h5disp

Purpose

Display contents of HDF5 file

Syntax

```
h5disp(filename)
h5disp(filename,location)
h5disp(filename,location,mode)
```

Description

`h5disp(filename)` displays the structure (metadata) of the entire HDF5 file, `filename`.

`h5disp(filename,location)` displays the metadata for the specified location.

`h5disp(filename,location,mode)` displays the file metadata according to the value of `mode`.

Input Arguments

filename

Text string specifying the name of an HDF5 file.

location

Text string specifying the full path to a location in an HDF5 file. To display the metadata for the entire file, specify `'/'` as the value of `location`. If `location` is a group, `h5disp` displays all objects below the group.

mode

Either of the following text strings.

Value	Description
`min`	Minimal, display only group and data set names.
`simple`	Display data set metadata and attribute values, if the attribute is an integer, floating point, or a scalar string.

Default: `simple`

h5disp

Examples

Display the entire contents of an HDF5 file.

```
h5disp('example.h5')
```

Display metadata for one data set in an HDF5 file.

```
h5disp('example.h5','/g4/world');
```

See Also

`h5info`

Tutorials

- "Using the High-Level HDF5 Functions to Import Data"

Purpose

Return information about HDF5 file

Syntax

```
info = h5info(filename)
info = h5info(filename,location)
```

Description

`info = h5info(filename)` returns information about the entire HDF5 file, specified by `filename`.

`info = h5info(filename,location)` returns information about the group, data set, or named datatype specified by `location` in the HDF5 file, `filename`.

Input Arguments

filename

Text string specifying the name of an HDF5 file.

location

Text string specifying the location of a group, data set, or named datatype in an HDF5 file.

Output Arguments

info

A structure containing information about the HDF5 file. The set of fields in the structure depends on the `location` parameter. The first field is always `'Filename'`. Other fields that might be present in the `info` structure are as follows.

Location Type	Field	Description
Files and Groups		
	`Name`	Text string specifying name of the group. If you specify only a file name, this value is `'/'`.

Location Type	Field	Description
	`Groups`	Array of structures describing subgroups.
	`Datasets`	Array of structures describing data sets.
	`Datatypes`	Array of structures describing named datatypes.
	`Links`	Array of structures describing soft, external, user-defined, and certain hard links.
	`Attributes`	Array of structures describing group attributes.
Data sets		
	`Name`	Text string specifying the name of the data set.
	`Datatype`	Structure describing the datatype.
	`Dataspace`	Structure describing the size of the dataset.
	`ChunkSize`	Extents of the data set's chunk size, if defined.
	`FillValue`	Data set's fill value, if defined.
	`Filter`	Array of structures describing any defined filters such as compression.
	`Attributes`	Array of structures describing data set attributes.
Named Datatypes		

Location Type	Field	Description
	`Name`	Text string specifying the name of the datatype object.
	`Class`	HDF5 class of the named datatype.
	`Type`	Text string or struct further describing the datatype.
	`Size`	Size of the named datatype in bytes.

Examples

Return all information.

```
info = h5info('example.h5');
```

Return information about a group and all data sets contained within the group.

```
info = h5info('example.h5','/g4');
```

Return information about a specific dataset.

```
info = h5info('example.h5','/g4/time');
```

See Also

`h5disp`

Tutorials

- “Using the High-Level HDF5 Functions to Import Data”

h5read

Purpose

Read data from HDF5 data set

Syntax

```
data = h5read(filename,datasetname)
data = h5read(filename,datasetname,start,count)
data = h5read(filename,datasetname,start,count,stride)
```

Description

`data = h5read(filename,datasetname)` retrieves all of the data from the HDF5 data set `datasetname` in the file `filename`.

`data = h5read(filename,datasetname,start,count)` reads a subset of data from the data set `datasetname`. `start` is the one-based index of the first element to be read. `count` defines how many elements to read along each dimension. If a particular element of `count` is `Inf`, `h5read` reads data until the end of the corresponding dimension.

`data = h5read(filename,datasetname,start,count,stride)` reads a subset of data, where `stride` specifies the interelement spacing along each data set extent.

Input Arguments

filename

Text string specifying the name of an HDF5 file.

datasetname

Text string specifying the name of a data set in an HDF5 file.

start

Numeric index value specifying the place to start reading data in the dataset in an HDF5 file. Indices are 1-based.

count

Numeric value specifying the amount of data to read.

stride

Numeric value specifying the intervalue spacing during the read operation. For example, a spacing of 2 indicates reading every other value.

Output Arguments

data

Data read from the data set.

Examples

Read an entire data set.

```
h5disp('example.h5','/g4/lat');
data = h5read('example.h5','/g4/lat');
```

Read the first 5-by-3 subset of a data set.

```
h5disp('example.h5','/g4/world');
data = h5read('example.h5','/g4/world',[1 1],[5 3]);
```

Read a data set of references to other data sets.

```
h5disp('example.h5','/g3/reference');
data = h5read('example.h5','/g3/reference');
```

See Also

`h5write` | `h5readatt` | `h5disp` | `h5writeatt`

Tutorials

- "Using the High-Level HDF5 Functions to Import Data"

h5readatt

Purpose

Read attribute from HDF5 file

Syntax

```
attval = h5readatt(filename,location,attr)
```

Description

`attval = h5readatt(filename,location,attr)` retrieves the value for the named attribute `attr` from the given `location` in the HDF5 file `filename`.

Input Arguments

filename

Text string specifying the name of an HDF5 file.

location

Text string specifying the full path of the attribute in an HDF5 file. `location` can refer to either a group or a data set.

attr

Text string specifying the name of an attribute in an HDF5 file.

Output Arguments

attval

Value of the attribute.

Examples

Read a group attribute.

```
attval = h5readatt('example.h5','/','attr2');
```

Read a data set attribute.

```
attval = h5readatt('example.h5','/g4/lon','units');
```

See Also

`h5writeatt | h5info`

Tutorials

- “Using the High-Level HDF5 Functions to Import Data”

Purpose

Write to HDF5 data set

Syntax

```
h5write(filename,datasetname,data)
h5write(filename,datasetname,data,start,count)
h5write(filename,datasetname,data,start,count,stride)
```

Description

`h5write(filename,datasetname,data)` writes `data` to an entire data set, `datasetname`, in the HDF5 file, `filename`.

`h5write(filename,datasetname,data,start,count)` writes a subset of the `data` to a data set, `datasetname`, in the HDF5 file, `filename`. `start` is a one-based index value that specifies the first element to be written. `count` specifies the number of elements to write along each dimension. `h5write` extends an extendable data set along any unlimited dimensions, if necessary.

`h5write(filename,datasetname,data,start,count,stride)` writes a hyperslab of data, where `stride` specifies the inter-element spacing along each dimension.

Input Arguments

filename

Text string specifying the name of an HDF5 file.

datasetname

Text string specifying the name of a data set in the HDF5 file.

data

Data to be written to the HDF5 file. You can specify only floating-point and integer data sets.

start

Numeric index value specifying where in the data set to start writing to the file.

count

Numeric value specifying how much data to write to the file.

stride

Numeric value specifying the interelement spacing of data to write to the file.

Default: Vector of ones.

Definitions

Hyperslab

A hyperslab is a collection of points in a data space. The points can be contiguous or form a regular pattern of points or blocks in a data space.

Examples

Write to an entire data set.

```
h5create('myfile.h5','/DS1',[10 20]);
mydata = rand(10,20);
h5write('myfile.h5', '/DS1', mydata);
```

Write a hyperslab of data to the last 5-by-7 block of a data set.

```
 h5create('myfile.h5','/DS2',[10 20]);
        mydata = rand(5,7);
        h5write('myfile.h5','/DS2',mydata,[6 14],[5 7]);
```

Append data to an unlimited data set.

```
h5create('myfile.h5','/DS3',[20 Inf],'ChunkSize',[5 5]);
for j = 1:10
      data = j*ones(20,1);
      start = [1 j];
      count = [20 1];
      h5write('myfile.h5','/DS3',data,start,count);
end
```

```
h5disp('myfile.h5');
```

Limitations

- `h5write` supports only floating point and integer data sets. To write to string data sets, you must use the H5D package.

See Also

`h5read` | `h5create` | `h5writeatt` | `h5disp` | `H5D.create` | `H5D.write`

Tutorials

- "Exporting to Hierarchical Data Format (HDF5) Files"

h5writeatt

Purpose

Write HDF5 attribute

Syntax

`h5writeatt(filename,location,attname,attvalue)`

Description

`h5writeatt(filename,location,attname,attvalue)` writes the attribute named `attname` with the value `attvalue` to the HDF5 file `filename`. The parent object `location` can be either a group or variable. `location` is the complete path name of the group or variable to which you want to associate the attribute.

Input Arguments

filename

Text string specifying the name of an HDF5 file.

location

Text string specifying the full path identifying a group or variable in an HDF5 file.

attname

Text string specifying the name of an attribute in an HDF5 file. If the attribute does not exist, `h5writeatt` creates the attribute with the name specified.

If the specified attribute already exists but does not have a datatype or dataspace consistent with `attvalue`, `h5writeatt` deletes the attribute and recreates it. String attributes are created with a scalar dataspace.

attvalue

Value to be written to the attribute in an HDF5 file.

Examples

Create a root group attribute whose value is the current time.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
h5writeatt('myfile.h5','/','creation_date',datestr(now));
```

Create a double-precision data set attribute.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5'
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
attData = [0 1 2 3];
h5writeatt('myfile.h5','/g4/world','attr',attData);
h5disp('myfile.h5','/g4/world');
```

See Also

h5readatt | h5disp | h5write

Tutorials

- "Exporting to Hierarchical Data Format (HDF5) Files"

H5.close

Purpose Close HDF5 library

Syntax `H5.close()`

Description `H5.close()` closes the HDF5 library.

See Also `H5.open`

Purpose Free unused memory in HDF5 library

Syntax

```
H5.garbage_collect()
```

Description `H5.garbage_collect()` frees unused memory in the HDF5 library.

H5.get_libversion

Purpose Version of HDF5 library

Syntax

```
[majnum,minnum,relnum] = H5.get_libversion()
```

Description `[majnum,minnum,relnum] = H5.get_libversion()` returns the version of the HDF5 library in use.

Purpose Open HDF5 library

Syntax `H5.open()`

Description `H5.open()` opens the HDF5 library.

See Also `H5.close`

H5.set_free_list_limits

Purpose Set size limits on free lists

Syntax

```
H5.set_free_list_limits(reg_global_lim,reg_list_lim,
   arr_global_lim,arr_list_lim,blk_global_lim,blk_list_lim)
```

Description

`H5.set_free_list_limits(reg_global_lim,reg_list_lim,arr_global_lim,arr_li`
sets size limits on all types of free lists.

Purpose

Close specified attribute

Syntax

```
H5A.close(attr_id)
```

Description

`H5A.close(attr_id)` terminates access to the attribute specified by `attr_id`, releasing the identifier.

See Also

`H5A.open`

H5A.create

Purpose

Create attribute

Syntax

```
attr_id = H5A.create(loc_id,name,type_id,space_id,acpl_id)
attr_id = H5A.create(loc_id,name,type_id,space_id,acpl_id,
   aapl_id)
```

Description

`attr_id = H5A.create(loc_id,name,type_id,space_id,acpl_id)` creates the attribute name that is attached to the object specified by `loc_id`. `loc_id` is a group, dataset, or named datatype identifier. The datatype and dataspace identifiers of the attribute, `type_id` and `space_id`, respectively, are created with the H5T and H5S interfaces. The attribute property list, `acpl_id`, is currently unused and should be set to `'H5P_DEFAULT'`. This interface corresponds to the 1.6.x version of `H5Acreate`.

`attr_id = H5A.create(loc_id,name,type_id,space_id,acpl_id,aapl_id)` creates the attribute with the additional attribute access property list identifier `aapl_id`. `aapl_id` should currently be set to `'H5P_DEFAULT'`. This interface corresponds to the 1.8.x version of `H5Acreate`.

Examples

```
acpl_id = H5P.create('H5P_ATTRIBUTE_CREATE');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
space_id = H5S.create('H5S_SCALAR');
fid = H5F.create('myfile.h5');
attr_id = H5A.create(fid,'my_attr',type_id,space_id,acpl_id);
H5A.close(attr_id);
H5F.close(fid);
```

See Also

`H5A.close` | `H5P.create`

Purpose

Delete attribute

Syntax

```
H5A.delete(loc_id,name)
```

Description

`H5A.delete(loc_id,name)` removes the attribute specified by `name` from the dataset, group, or named datatype specified by `loc_id`.

Examples

Delete a root group attribute.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5')
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR','H5P_DEFAULT');
gid = H5G.open(fid,'/');
H5A.delete(gid,'attr1');
H5G.close(gid);
H5F.close(fid);
```

H5A.get_info

Purpose

Information about attribute

Syntax

```
info = H5A.get_info(attr_id)
```

Description

`info = H5A.get_info(attr_id)` returns information about an attribute specified by `attr_id`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/');
attr_id = H5A.open(gid,'attr1');
info = H5A.get_info(attr_id);
H5A.close(attr_id);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.open`

Purpose

Attribute name

Syntax

```
attr_name = H5A.get_name(attr_id)
```

Description

`attr_name = H5A.get_name(attr_id)` returns the name of the attribute specified by `attr_id`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g1/g1.1');
idx_type = 'H5_INDEX_NAME';
order = 'H5_ITER_INC';
attr_id = H5A.open_by_idx(gid,'dset1.1.1',idx_type,order,0);
name = H5A.get_name(attr_id);
H5A.close(attr_id);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.open_by_idx`

H5A.get_space

Purpose

Copy of attribute data space

Syntax

```
dspace_id = H5A.get_space(attr_id)
```

Description

`dspace_id = H5A.get_space(attr_id)` returns a copy of the data space for the attribute specified by `attr_id`.

Examples

Retrieve the dimensions of an attribute data space.

```
fid = H5F.open('example.h5');
attr_id = H5A.open(fid,'attr2');
space = H5A.get_space(attr_id);
[~,dims] = H5S.get_simple_extent_dims(space);
H5A.close(attr_id);
H5F.close(fid);
```

See Also

`H5A.open` | `H5S.close`

Purpose

Copy of attribute data type

Syntax

```
type_id = H5A.get_type(attr_id)
```

Description

`type_id = H5A.get_type(attr_id)` returns a copy of the data type for the attribute specified by `attr_id`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/');
attr_id = H5A.open(gid,'attr1');
type_id = H5A.get_type(attr_id);
H5T.close(type_id);
H5A.close(attr_id);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.open` | `H5T.close`

H5A.iterate

Purpose

Execute function for attributes attached to object

Syntax

```
[status,idx_stop,cdata_out] = H5A.iterate(obj_id,idx_type,
   order,idx_start,iter_func,cdata_in)
H5A.iterate(loc_id,attr_idx,iterator_func)
```

Description

`[status,idx_stop,cdata_out] = H5A.iterate(obj_id,idx_type,order,idx_start,iter_func,cdata_in)` executes the specified function `iter_func` for each attribute connected to an object. `obj_id` identifies the object to which attributes are attached. `idx_type` is the type of index and valid values include the following.

'H5_INDEX_NAME'	An alpha-numeric index by attribute name
'H5_INDEX_CRT_ORDER'	An index by creation order

`order` specifies the index traversal order. Valid values include the following.

'H5_ITER_INC'	Iteration from beginning to end
'H5_ITER_DEC'	Iteration from end to beginning
'H5_ITER_NATIVE'	Iteration in the fastest available order

`idx_start` specifies the starting point of the iteration. `idx_stop` returns the point at which iteration was stopped. This allows an interrupted iteration to be resumed.

The callback function, `iter_func`, must have the following signature:

```
[status,cdata_out] =
iter_func(obj_id,attr_name,info,cdata_in)
```

`cdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `cdata_in` parameter. The `cdata_out` of an iteration step forms the `cdata_in` for the next iteration

step. Then, the final `cdata_out` at the end of the iteration is returned to the caller as `cdata_out`. This form of `H5A.iterate` corresponds to the `H5Aiterate2` function in the HDF5 C API.

`status` value returned by `iter_func` is interpreted as follows.

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

`H5A.iterate(loc_id,attr_idx,iterator_func)` executes the specified function for each attribute of the group, dataset, or named datatype specified by `loc_id`. The `attr_idx` argument specifies where the iteration begins. `iterator_func` must be a function handle.

The iterator function must have the following signature:

```
status = iterator_func(loc_id,attr_name)
```

`loc_id` still specifies the group, dataset, or named data type passed into `H5A.iterate`, and `attr_name` specifies the current attribute. This form of `H5A.iterate` corresponds to `H5Aiterate1` function in the HDF5 C API.

H5A.open

Purpose

Open attribute

Syntax

```
attr_id = H5A.open(obj_id,attr_name)
attr_id = H5A.open(obj_id,attr_name,aapl_id)
```

Description

`attr_id = H5A.open(obj_id,attr_name)` opens an attribute for an object specified by a parent object identifier and attribute name.

`attr_id = H5A.open(obj_id,attr_name,aapl_id)` opens an attribute with an attribute access property list identifier, `aapl_id`. The only currently valid value for `aapl_id` is `'H5P_DEFAULT'`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/');
attr_id = H5A.open(gid,'attr1');
H5A.close(attr_id);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.close` | `H5A.open_by_name` | `H5A.open_by_idx`

Purpose

Open attribute specified by index

Syntax

```
attr_id = H5A.open_by_idx(loc_id,obj_name,idx_type,order,n)
attr_id = H5A.open_by_idx(loc_id,obj_name,idx_type,order,n,
   aapl_id,lapl_id)
```

Description

`attr_id = H5A.open_by_idx(loc_id,obj_name,idx_type,order,n)` opens an existing attribute at index `n` attached to an object specified by its location, `loc_id`, and name, `obj_name`.

`idx_type` is the type of index and valid values include the following.

'H5_INDEX_NAME'	An alpha-numeric index by attribute name
'H5_INDEX_CRT_ORDER'	An index by creation order

`order` specifies the index traversal order. Valid values include the following.

'H5_ITER_INC'	Iteration from beginning to end
'H5_ITER_DEC'	Iteration from end to beginning
'H5_ITER_NATIVE'	Iteration in the fastest available order

`attr_id = H5A.open_by_idx(loc_id,obj_name,idx_type,order,n,aapl_id,lapl_id)` opens an attribute with attribute access property list, `aapl_id`, and link access property list, `lapl_id`. The `aapl_id` argument must currently be specified as `'H5P_DEFAULT'`. Also, `lapl_id` can be specified by `'H5P_DEFAULT'`.

Examples

Loop through a set of dataset attributes in reverse alphabetical order.

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g1/g1.1');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
```

```
info = H5O.get_info(dset_id);
for idx = 0:info.num_attrs-1
  attr_id =H5A.open_by_idx(gid,'dset1.1.1','H5_INDEX_NAME','H5_ITER_DEC',
  fprintf('attribute name:  %s\n',H5A.get_name(attr_id));
  H5A.close(attr_id);
end
H5G.close(gid);
H5F.close(fid);
```

See Also

H5A.open | H5A.open_by_name | H5A.close

Purpose

Open attribute specified by name

Syntax

```
attr_id = H5A.open_by_name(loc_id,obj_name,attr_name)
attr_id = H5A.open_by_name(loc_id,obj_name,attr_name,aapl_id,
   lapl_id)
```

Description

`attr_id = H5A.open_by_name(loc_id,obj_name,attr_name)` opens an existing attribute `attr_name` attached to an object specified by its location `loc_id` and name `obj_name`.

`attr_id = H5A.open_by_name(loc_id,obj_name,attr_name,aapl_id,lapl_id)` opens an existing attribute with the attribute access property list `aapl_id` and link access property list `lacpl_id`. `aapl_id` must be specified as `'H5P_DEFAULT'`. `lapl_id` may also be specified by `'H5P_DEFAULT'`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g1/g1.1');
attr_id = H5A.open_by_name(gid,'dset1.1.1','attr1');
H5A.close(attr_id);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.close` | `H5A.open` | `H5A.open_by_idx`

H5A.read

Purpose

Read attribute

Syntax

```
attr = H5A.read(attr_id)
attr = H5A.read(attr_id, mem_type_id)
```

Description

`attr = H5A.read(attr_id)` reads the attribute specified by `attr_id`. MATLAB will determine the appropriate memory datatype.

`attr = H5A.read(attr_id, mem_type_id)` reads the attribute specified by `attr_id`. `mem_type_id` specifies the attribute's memory datatype and should usually be given as `'H5ML_DEFAULT'`, which specifies that MATLAB will determine the appropriate memory datatype.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. If the HDF5 library reports the attribute size as 3-by-4-by-5, then the corresponding MATLAB array size is 5-by-4-by-3. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/');
attr_id = H5A.open(gid,'attr1');
data = H5A.read(attr_id);
H5A.close(attr_id);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.open | H5A.write`

Purpose

Write attribute

Syntax

```
H5A.write(attr_id,type_id,buf)
```

Description

`H5A.write(attr_id,type_id,buf)` writes the data in buf into the attribute specified by `attr_id`. `type_id` specifies the attribute's memory datatype. The memory datatype should be `'H5ML_DEFAULT'`, which specifies that MATLAB should determine the appropriate memory datatype.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. If the MATLAB array size is 5-by-4-by-3, then the HDF5 library should be reporting the attribute size as 3-by-4-by-5. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Write a scalar double precision attribute.

```
acpl = H5P.create('H5P_ATTRIBUTE_CREATE');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
space_id = H5S.create('H5S_SCALAR');
fid = H5F.create('myfile.h5');
attr_id = H5A.create(fid,'my_attr',type_id,space_id,acpl);
H5A.write(attr_id,'H5ML_DEFAULT',10.0)
H5A.close(attr_id);
H5F.close(fid);
H5T.close(type_id);
```

See Also

`H5A.read`

H5D.close

Purpose

Close dataset

Syntax

`H5D.close(dataset_id)`

Description

`H5D.close(dataset_id)` ends access to a dataset specified by `dataset_id` and releases resources used by it.

See Also

`H5D.create` | `H5D.open`

Purpose

Create new dataset

Syntax

```
dataset_id = H5D.create(loc_id,name,type_id,space_id,
   plist_id)
dataset_id = H5D.create(loc_id,name,type_id,space_id,lcpl_id,
   dcpl_id,dapl_id)
```

Description

`dataset_id = H5D.create(loc_id,name,type_id,space_id,plist_id)` creates the data set specified by `name` in the file or in the group specified by `loc_id`. `type_id` and `space_id` identify the datatype and dataspace, respectively. `plist_id` identifies the dataset creation property list. This interface corresponds to the `H5Dcreate1` function in the HDF5 library C 1.6 API.

`dataset_id = H5D.create(loc_id,name,type_id,space_id,lcpl_id,dcpl_id,dapl_id)` creates the data set with three distinct property lists:

`lcpl_id`	link creation property list
`dcpl_id`	dataset creation property list
`dapl_id`	dataset access property list

This interface corresponds to the `H5Dcreate` function in the HDF5 library C 1.8 API.

Examples

Create a 10x5 double precision dataset with default property list settings.

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [10 5];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = 'H5P_DEFAULT';
```

```
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5S.close(space_id);
H5T.close(type_id);
H5D.close(dset_id);
H5F.close(fid);
h5disp('myfile.h5');
```

Create a 6x3 fixed length string dataset. Each string will have a length of 4 characters.

```
fid = H5F.create('myfile_strings.h5');
type_id = H5T.copy('H5T_C_S1');
H5T.set_size(type_id,4);
dims = [6 3];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = 'H5P_DEFAULT';
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5S.close(space_id);
H5T.close(type_id);
H5D.close(dset_id);
H5F.close(fid);
h5disp('myfile_strings.h5');
```

See Also

H5D.close | H5S.create_simple | H5S.close | H5T.copy

Purpose

Copy of dataset access property list

Syntax

```
plist_id = H5D.get_access_plist(dataset_id)
```

Description

`plist_id = H5D.get_access_plist(dataset_id)` returns a copy of the dataset access property list used to open the specified dataset.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
dapl = H5D.get_access_plist(dset_id);
H5P.close(dapl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5D.get_create_plist` | `H5P.close`

H5D.get_create_plist

Purpose

Copy of dataset creation property list

Syntax

```
plist_id = H5D.get_create_plist(dataset_id)
```

Description

`plist_id = H5D.get_create_plist(dataset_id)` returns the identifier to a copy of the dataset creation property list for the dataset specified by `dataset_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
dcpl = H5D.get_create_plist(dset_id);
H5P.close(dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5D.get_access_plist` | `H5P.close`

H5D.get_offset

Purpose Location of dataset in file

Syntax

```
offset = H5D.get_offset(dataset_id)
```

Description

`offset = H5D.get_offset(dataset_id)` returns the location in the file of the dataset specified by `dataset_id`. The location is expressed as an offset, in bytes, from the beginning of the file.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
offset = H5D.get_offset(dset_id);
H5D.close(dset_id);
H5F.close(fid);
```

H5D.get_space

Purpose

Copy of dataset data space

Syntax

```
dspace_id = H5D.get_space(dataset_id)
```

Description

`dspace_id = H5D.get_space(dataset_id)` returns an identifier for a copy of the data space for a dataset.

Examples

Retrieve the dimensions of an attribute data space.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
space = H5D.get_space(dset_id);
[~,dims] = H5S.get_simple_extent_dims(space);
H5S.close(space);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5D.open` | `H5S.close`

Purpose Determine if space is allocated

Syntax

```
status = H5D.get_space_status(dataset_id)
```

Description

`status = H5D.get_space_status(dataset_id)` determines whether space has been allocated for the dataset specified by `dataset_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
status = H5D.get_space_status(dset_id);
switch(status)
    case H5ML.get_constant_value('H5D_SPACE_STATUS_NOT_ALLOCATED')
        fprintf('Not allocated.\n');
    case H5ML.get_constant_value('H5D_SPACE_STATUS_ALLOCATED')
        fprintf('Allocated.\n');
    case H5ML.get_constant_value('H5D_SPACE_STATUS_PART_ALLOCATED')
        fprintf('Part allocated.\n');
end
H5D.close(dset_id);
H5F.close(fid);
```

See Also `H5D.get_space`

H5D.get_storage_size

Purpose Determine required storage size

Syntax

```
dataset_size = H5D.get_storage_size(dataset_id)
```

Description

`dataset_size = H5D.get_storage_size(dataset_id)` returns the amount of storage that is required for the dataset specified by `dataset_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
dataset_size = H5D.get_storage_size(dset_id);
H5D.close(dset_id);
H5F.close(fid);
```

H5D.get_type

Purpose

Copy of datatype

Syntax

```
type_id = H5D.get_type(dataset_id)
```

Description

`type_id = H5D.get_type(dataset_id)` returns an identifier for a copy of the data type for the dataset specified by `dataset_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
type_id = H5D.get_type(dset_id);
H5T.close(type_id);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

H5T.close

H5D.open

Purpose

Open specified dataset

Syntax

```
dataset_id = H5D.open(loc_id,name)
dataset_id = H5D.open(loc_id,name,dapl_id)
```

Description

`dataset_id = H5D.open(loc_id,name)` opens the dataset specified by `name` in the file or group specified by `loc_id`.

`dataset_id = H5D.open(loc_id,name,dapl_id)` opens the dataset specified by `name` in the file or group specified by `loc_id`. The dataset access property list, `dapl_id`, provides information regarding access to the dataset.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g2');
dset_id = H5D.open(gid,'dset2.2');
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5D.close`

Purpose

Read data from HDF5 dataset

Syntax

```
data = H5D.read(dataset_id)
data = H5D.read(dataset_id,mem_type_id,mem_space_id,
   file_space_id,dxpl)
```

Description

`data = H5D.read(dataset_id)` reads the entire dataset specified by `dataset_id`.

`data = H5D.read(dataset_id,mem_type_id,mem_space_id,file_space_id,dxpl)` reads the dataset specified by `dataset_id`. The `mem_type_id` input specifies the memory data type and should usually be `'H5ML_DEFAULT'` to allow MATLAB to determine the appropriate value. `mem_space_id` describes how the data is to be arranged in memory and should usually be `'H5S_ALL'`. The `file_space_id` input describes how the data is to be selected from the file. It also can be given as `'H5S_ALL'`, but this results in the entire dataset being read into memory. `dxpl` is the dataset transfer property list identifier and should usually be `'H5P_DEFAULT'`.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Read an entire dataset.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
data = H5D.read(dset_id);
H5D.close(dset_id);
H5F.close(fid);
```

Read the 2x3 hyperslab starting in the 4th row and 5th column of the example dataset.

```
plist = 'H5P_DEFAULT';
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g1/g1.1/dset1.1.1');
dims = fliplr([2 3]);
mem_space_id = H5S.create_simple(2,dims,[]);
file_space_id = H5D.get_space(dset_id);
offset = fliplr([3 4]);
block = fliplr([2 3]);
H5S.select_hyperslab(file_space_id,'H5S_SELECT_SET',offset,[],[],block);
data = H5D.read(dset_id,'H5ML_DEFAULT',mem_space_id,file_space_id,plist);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

H5D.open | H5D.write | H5S.create_simple

Purpose Change size of dataset dimensions

Syntax `H5D.set_extent(dset_id,h5_extents)`

Description `H5D.set_extent(dset_id,h5_extents)` changes the dimensions of the dataset `dset_id` to the sizes specified in `h5_extents`.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_extents` parameter assumes C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples Extend an unlimited one-dimensional dataset from a length of 10 to a length of 20.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5')
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR','H5P_DEFAULT');
dset_id = H5D.open(fid,'/g4/time');
H5D.set_extent(dset_id,20);
H5D.close(dset_id);
H5F.close(fid);
```

H5D.vlen_get_buf_size

Purpose

Determine variable length storage requirements

Syntax

```
size = H5D.vlen_get_buf_size(dataset_id,type_id,space_id)
```

Description

`size = H5D.vlen_get_buf_size(dataset_id,type_id,space_id)` determines the number of bytes required to store the VL data from the dataset, using the `space_id` for the selection in the dataset on disk and the `type_id` for the memory representation of the VL data in memory.

Purpose

Write data to HDF5 dataset

Syntax

```
H5D.write(dataset_id,mem_type_id,mem_space_id,file_space_id,
   plist_id,buf)
```

Description

`H5D.write(dataset_id,mem_type_id,mem_space_id,file_space_id,plist_id,b` writes the dataset specified by `dataset_id` from the application memory buffer `buf` into the file. `plist_id` specifies the data transfer properties. `mem_type_id` identifies the memory datatype of the dataset. `mem_space_id` and `file_space_id` define the part of the dataset to write. The memory datatype should usually be `'H5ML_DEFAULT'`, which specifies that MATLAB should determine the appropriate memory datatype.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Write to the entire 36-by-19 `/g4/world` example dataset.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5')
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
plist = 'H5P_DEFAULT';
fid = H5F.open('myfile.h5','H5F_ACC_RDWR',plist);
dset_id = H5D.open(fid,'/g4/world');
dims = [36 19];
data = rand(dims);
H5D.write(dset_id,'H5ML_DEFAULT','H5S_ALL','H5S_ALL',plist,data);
H5D.close(dset_id);
H5F.close(fid);
```

Write to the entire two-element /g3/VLstring dataset.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
h5disp('myfile.h5','/g3/VLstring');
plist = 'H5P_DEFAULT';
fid = H5F.open('myfile.h5','H5F_ACC_RDWR',plist);
dset_id = H5D.open(fid,'/g3/VLstring');
data = {'dogs'; 'dogs and cats'};
H5D.write(dset_id,'H5ML_DEFAULT','H5S_ALL','H5S_ALL',plist,data);
H5D.close(dset_id);
H5F.close(fid);
data_out = h5read('myfile.h5','/g3/VLstring');
```

Write a 10-by-5 block of data to the location starting at row index 15 and column index 5 of the same dataset. Recall that indexing is zero-based.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
plist = 'H5P_DEFAULT';
fid = H5F.open('myfile.h5','H5F_ACC_RDWR',plist);
dset_id = H5D.open(fid,'/g4/world');
start = [15 5];
h5_start = fliplr(start);
block = [10 5];
h5_block = fliplr(block);
mem_space_id = H5S.create_simple(2,h5_block,[]);
file_space_id = H5D.get_space(dset_id);
H5S.select_hyperslab(file_space_id,'H5S_SELECT_SET',h5_start,[],[],h5_blo
data = rand(block);
H5D.write(dset_id,'H5ML_DEFAULT',mem_space_id,file_space_id,plist,data);
H5D.close(dset_id);
H5F.close(fid);
```

See Also `H5D.read`

H5DS.attach_scale

Purpose

Attach dimension scale to specific dataset dimension

Syntax

```
H5DS.attach_scale(dataset_id,dimscale_id,idx)
```

Description

`H5DS.attach_scale(dataset_id,dimscale_id,idx)` attaches a dimension scale `dimscale_id` to dimension `idx` of the dataset `dataset_id`.

Note The ordering of the dimension scale indices are the same as the HDF5 library C API.

Examples

Add the `'lon'` and `'lat'` dimension scales to the `'world'` dataset.

```
plist = 'H5P_DEFAULT';
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR',plist);
world_dset_id = H5D.open(fid,'/g4/world',plist);
lat_dset_id = H5D.open(fid,'/g4/lat',plist);
lon_dset_id = H5D.open(fid,'/g4/lon',plist);
H5DS.attach_scale(world_dset_id,lat_dset_id,0);
H5DS.attach_scale(world_dset_id,lon_dset_id,1);
H5D.close(lat_dset_id);
H5D.close(lon_dset_id);
H5D.close(world_dset_id);
H5F.close(fid);
```

See Also

`H5DS.detach_scale`

H5DS.detach_scale

Purpose Detach dimension scale from specific dataset dimension

Syntax

```
H5DS.detach_scale(dataset_id,dimscale_id,idx)
```

Description `H5DS.detach_scale(dataset_id,dimscale_id,idx)` detaches dimension scale `dimscale_id` from dimension `idx` of the dataset `dataset_id`.

Note The ordering of the dimension scale indices are the same as the HDF5 library C API.

See Also `H5DS.attach_scale`

H5DS.get_label

Purpose

Retrieve label from specific dataset dimension

Syntax

```
label = H5DS.get_label(dataset_id,idx)
```

Description

`label = H5DS.get_label(dataset_id,idx)` retrieves the label for dimension `idx` of the dataset `dataset_id`.

Note The ordering of the dimension scale indices are the same as the HDF5 library C API.

Examples

```
fid = H5F.open('example.h5');
world_dset_id = H5D.open(fid,'/g4/world');
label = H5DS.get_label(world_dset_id,0);
H5D.close(world_dset_id);
H5F.close(fid);
```

See Also

`H5DS.set_label`

Purpose Number of scales attached to dataset dimension

Syntax

```
num_scales = H5DS.get_num_scales(dataset_id,idx)
```

Description `num_scales = H5DS.get_num_scales(dataset_id,idx)` determines the number of dimension scales that are attached to dimension `idx` of the dataset `dataset_id`.

Examples

```
fid = H5F.open('example.h5');
world_dset_id = H5D.open(fid,'/g4/world');
num_scales = H5DS.get_num_scales(world_dset_id,0);
H5D.close(world_dset_id);
H5F.close(fid);
```

H5DS.get_scale_name

Purpose Name of dimension scale

Syntax

```
name = H5DS.get_scale_name(dimscale_id)
```

Description `name = H5DS.get_scale_name(dimscale_id)` retrieves the name of the dimension scale `dimscale_id`.

Examples

```
fid = H5F.open('example.h5');
lat_dset_id = H5D.open(fid,'/g4/lat');
scale_name = H5DS.get_scale_name(lat_dset_id);
H5D.close(lat_dset_id);
H5F.close(fid);
```

See Also `H5DS.set_scale`

H5DS.is_scale

Purpose

Determine if dataset is a dimension scale

Syntax

```
bool = H5DS.is_scale(dataset_id)
```

Description

`bool = H5DS.is_scale(dataset_id)` determines whether the dataset `dataset_id` is a dimension scale.

Examples

```
fid = H5F.open('example.h5');
lat_dset_id = H5D.open(fid,'/g4/lat');
if H5DS.is_scale(lat_dset_id)
    fprintf('/g4/lat is a dimension scale.\n');
else
    fprintf('/g4/lat is not a dimension scale.\n');
end
H5D.close(lat_dset_id);
H5F.close(fid);
```

H5DS.iterate_scales

Purpose

Iterate on scales attached to dataset dimension

Syntax

```
[status,idx_out,opdata_out] = H5DS.iterate_scales(dset_id,dim,
   idx_in,iter_func,opdata_in)
```

Description

`[status,idx_out,opdata_out] = H5DS.iterate_scales(dset_id,dim,idx_in,iter_func,opdata_in)` iterates over the scales attached to dimension `dim` of the dataset `dset_id` to perform a common operation whose function handle is `iter_func`.

`idx_in` specifies the starting point of the iteration. `idx_out` returns the point at which iteration was stopped. This allows an interrupted iteration to be resumed. If `idx_in` is `[]`, then the iterator starts at the first member.

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] =
iter_func(dset_id,dim,dimscale_id,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final `opdata_out` at the end of the iteration is then returned to the caller as `opdata_out`.

`dimscale_id` specifies the current dimension scale dataset identifier and `dim` is the associated dimension.

status value returned by `iter_func` is interpreted as follows:

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

H5DS.set_label

Purpose Set label for dataset dimension

Syntax `H5DS.set_label(dataset_id,idx,label)`

Description `H5DS.set_label(dataset_id,idx,label)` sets a label for dimension `idx` of the dataset `dataset_id`.

Note The ordering of the dimension scale indices are the same as the HDF5 library C API.

Examples

```
plist = 'H5P_DEFAULT';
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR',plist);
world_dset_id = H5D.open(fid,'/g4/world',plist);
H5DS.set_label(world_dset_id,0,'latitude');
H5DS.set_label(world_dset_id,1,'longitude');
H5D.close(world_dset_id);
H5F.close(fid);
```

See Also `H5DS.get_label`

H5DS.set_scale

Purpose Convert dataset to dimension scale

Syntax `H5DS.set_scale(dataset_id,dim_name)`

Description `H5DS.set_scale(dataset_id,dim_name)` converts the dataset, `dataset_id`, to a dimension scale with name `dim_name`.

Examples Create a dimension scale with name `'xdim'`. The dataset has the name, `'x'`.

```
fid = H5F.create('myfile.h5');
space_id = H5S.create_simple(1,10,10);
dtype = 'H5T_NATIVE_INT';
dcpl = 'H5P_DEFAULT';
dset_id = H5D.create(fid,'x',dtype,space_id,dcpl);
H5DS.set_scale(dset_id,'xdim');
H5S.close(space_id);
H5D.close(dset_id);
H5F.close(fid);
```

See Also `H5DS.get_scale_name`

H5E.clear

Purpose Clear error stack

Syntax

```
H5E.clear()
```

Description `H5E.clear()` clears the error stack for the current thread.

H5E.get_major

Purpose

Description of major error number

Syntax

```
err_string = H5E.get_major(major_number)
```

Description

`err_string = H5E.get_major(major_number)` returns a character string describing an error specified by the major error number, `major_number`.

The HDF5 group has deprecated the use of this function.

See Also

`H5E.get_minor`

H5E.get_minor

Purpose Description of minor error number

Syntax `err_string = H5E.get_minor(minor_number)`

Description `err_string = H5E.get_minor(minor_number)` returns a character string describing an error specified by the minor error number, `minor_number`.

The HDF5 group has deprecated the use of this function.

See Also `H5E.get_major`

Purpose Walk error stack

Syntax

```
H5E.walk(direction,func)
```

Description `H5E.walk(direction,func)` walks the error stack for the current thread and calls the specified function for each error along the way. `func` is a function handle. `direction` specifies how the error stack is traversed and can be given by one of the following strings or the numeric equivalent.

`'H5E_WALK_UPWARD'`

`'H5E_WALK_DOWNWARD'`

The specified function must have the following signature:

```
status = func(n,error_struct)
```

where `n` is the indexed position of the error in the stack and `error_struct` is a structure with the following fields:

`maj_num`	Major error number
`min_num`	Minor error number
`func_name`	Function in which the error occurred
`file_name`	File in which the error occurred
`line`	Line in file where error occurs
`desc`	Optional supplied description

This function corresponds to the `H5Ewalk1` function in the HDF5 library C API.

See Also `H5ML.get_constant_value`

H5F.close

Purpose Close HDF5 file

Syntax `H5F.close(file_id)`

Description `H5F.close(file_id)` terminates access to HDF5 file identified by `file_id`, flushing all data to storage.

See Also `H5F.open`

Purpose

Create HDF5 file

Syntax

```
file_id = H5F.create(filename)
file_id = H5F.create(name,flags,fcpl_id,fapl_id)
```

Description

`file_id = H5F.create(filename)` creates the file specified by `filename` with default library properties if the file does not already exist.

`file_id = H5F.create(name,flags,fcpl_id,fapl_id)` creates the file specified by `name`. `flags` specifies whether to truncate the file, if it already exists, or to fail if the file already exists. `flags` can be specified by one of the following strings or the numeric equivalent:

'H5F_ACC_TRUNC'	overwrite any existing file by the same name
'H5F_ACC_EXCL'	do not overwrite an existing file

`fcpl_id` is the file creation property list identifier. `fapl_id` is the file access property list identifier. A value of `'H5P_DEFAULT'` for either property list indicates that the library should use default values for the appropriate property list.

Examples

Create an HDF5 file called `'myfile.h5'`.

```
fid = H5F.create('myfile.h5');
H5F.close(fid);
```

Create an HDF5 file called `'myfile.h5'`, overwriting any existing file by the same name. Default file access and file creation properties shall apply.

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5F.close` | `H5P.create` | `H5ML.get_constant_value`

Purpose

Flush buffers to disk

Syntax

```
H5F.flush(object_id,scope)
```

Description

`H5F.flush(object_id,scope)` causes all buffers associated with a file to be immediately flushed to disk without removing the data from the cache. `object_id` can be any object associated with the file, including the file itself, a dataset, a group, an attribute, or a named data type. `scope` specifies whether the scope of the flushing action is global or local. `scope` may be one of the following strings:

`'H5F_SCOPE_GLOBAL'`

`'H5F_SCOPE_LOCAL'`

H5F.get_access_plist

Purpose File access property list

Syntax

```
fapl_id = H5F.get_access_plist(file_id)
```

Description `fapl_id = H5F.get_access_plist(file_id)` returns the file access property list identifier of the file specified by `file_id`.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
H5P.close(fapl);
H5F.close(fid);
```

See Also `H5F.get_create_plist`

Purpose

File creation property list

Syntax

```
fcpl_id = H5F.get_create_plist(file_id)
```

Description

`fcpl_id = H5F.get_create_plist(file_id)` returns a file creation property list identifier identifying the creation properties used to create the file specified by `file_id`.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
H5P.close(fcpl);
H5F.close(fid);
```

See Also

`H5F.get_access_plist`

H5F.get_filesize

Purpose

Size of HDF5 file

Syntax

```
size = H5F.get_filesize(file_id)
```

Description

`size = H5F.get_filesize(file_id)` returns the size of the HDF5 file specified by `file_id`.

Purpose

Amount of free space in file

Syntax

```
free_space = H5F.get_freespace(file_id)
```

Description

`free_space = H5F.get_freespace(file_id)` returns the amount of space that is unused by any objects in the file specified by `file_id`.

H5F.get_info

Purpose

Global information about file

Syntax

```
file_info = H5F.get_info(obj_id)
```

Description

`file_info = H5F.get_info(obj_id)` returns global information for the file associated with the object identifier `obj_id`. For details about the fields of the `file_info` structure, please refer to the HDF5 documentation.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'g2');
info = H5F.get_info(gid);
H5G.close(gid);
H5F.close(fid);
```

H5F.get_mdc_config

Purpose

Metadata cache configuration

Syntax

```
config_struct = H5F.get_mdc_config(file_id)
```

Description

`config_struct = H5F.get_mdc_config(file_id)` returns the current metadata cache configuration for the target file.

Examples

```
fid = H5F.open('example.h5');
config = H5F.get_mdc_config(fid);
H5F.close(fid);
```

See Also

`H5F.set_mdc_config`

H5F.get_mdc_hit_rate

Purpose Metadata cache hit-rate

Syntax

```
hitRate = H5F.get_mdc_hit_rate(file_id)
```

Description `hitRate = H5F.get_mdc_hit_rate(file_id)` queries the metadata cache of the target file to obtain its hit-rate since the last time hit-rate statistics were reset. If the cache has not been accessed since the last time the hit-rate statistics were reset, the hit-rate is defined to be 0.0. The hit-rate is calculated as

```
(cache hits / (cache hits + cache misses))
```

Examples

```
fid = H5F.open('example.h5');
hit_rate = H5F.get_mdc_hit_rate(fid);
H5F.close(fid);
```

See Also `H5F.get_mdc_config`

H5F.get_mdc_size

Purpose

Metadata cache size data

Syntax

```
[max_sz,min_clean_sz,cursz,
   num_cur_entries] = H5F.get_mdc_size(fileId)
```

Description

`[max_sz,min_clean_sz,cursz,num_cur_entries] = H5F.get_mdc_size(fileId)` queries the metadata cache of the target file to obtain current metadata cache size information.

Examples

```
fid = H5F.open('example.h5');
[maxsz,minsz,cursz,nent] = H5F.get_mdc_size(fid);
H5F.close(fid);
```

See Also

`H5F.get_mdc_config`

H5F.get_name

Purpose

Name of HDF5 file

Syntax

```
name = H5F.get_name(obj_id)
```

Description

`name = H5F.get_name(obj_id)` returns the name of the file to which the object `obj_id` belongs. The object can be a group, dataset, attribute, or named data type.

Examples

```
fid = H5F.open('example.h5');
name = H5F.get_name(fid);
H5F.close(fid);
```

See Also

`H5A.get_name` | `H5I.get_name`

H5F.get_obj_count

Purpose Number of open objects in HDF5 file

Syntax

```
obj_count = H5F.get_obj_count(file_id,types)
```

Description

`obj_count = H5F.get_obj_count(file_id,types)` returns the number of open object identifiers for the file specified by `file_id` for the specified type. `types` can be one of the following strings.

- 'H5F_OBJ_FILE'
- 'H5F_OBJ_DATASET'
- 'H5F_OBJ_GROUP'
- 'H5F_OBJ_DATATYPE'
- 'H5F_OBJ_ATTR'
- 'H5F_OBJ_ALL'
- 'H5F_OBJ_LOCAL'

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g2');
obj_count = H5F.get_obj_count(fid,'H5F_OBJ_GROUP');
H5G.close(gid);
H5F.close(fid);
```

See Also `H5F.get_obj_ids`

H5F.get_obj_ids

Purpose

List of open HDF5 file objects

Syntax

```
[num_obj_ids,obj_id_list] = H5F.get_obj_ids(file_id,types,
   max_objs)
```

Description

`[num_obj_ids,obj_id_list] = H5F.get_obj_ids(file_id,types,max_objs)` returns a list of all open identifiers for HDF5 objects of the type specified by `types` in the file specified by `file_id`. The `max_objs` input specifies the maximum number of object identifiers to return. `num_obj_ids` is the total number of objects in the list. `types` can be one of the following strings.

'H5F_OBJ_FILE'

'H5F_OBJ_DATASET'

'H5F_OBJ_GROUP'

'H5F_OBJ_DATATYPE'

'H5F_OBJ_ATTR'

'H5F_OBJ_ALL'

'H5F_OBJ_LOCAL'

Examples

```
fid = H5F.open('example.h5');
gid1 = H5G.open(fid,'/g1');
gid2 = H5G.open(fid,'/g2');
gid3 = H5G.open(fid,'/g3');
gid4 = H5G.open(fid,'/g4');
[num_obj_ids,objs] = H5F.get_obj_ids(fid,'H5F_OBJ_GROUP',3);
H5G.close(gid1);
H5G.close(gid2);
H5G.close(gid3);
H5G.close(gid4);
H5F.close(fid);
```

See Also

`H5F.get_obj_count`

H5F.is_hdf5

Purpose

Determine if file is HDF5

Syntax

```
value = H5F.is_hdf5(name)
```

Description

`value = H5F.is_hdf5(name)` returns a positive number if the file specified by `name` is in the HDF5 format, and zero if it is not. A negative return value indicates failure.

Examples

```
value = H5F.is_hdf5('example.tif');
if value > 0
    fprintf('example.tif is an HDF5 file\n');
else
    fprintf('example.tif is not an HDF5 file\n');
end
```

H5F.mount

Purpose

Mount HDF5 file onto specified location

Syntax

```
H5F.mount(loc_id,name,child_id,plist_id)
```

Description

`H5F.mount(loc_id,name,child_id,plist_id)` mounts the file specified by `child_id` onto the group specified by `loc_id` and `name`, using the mount properties specified by `plist_id`.

Examples

Mount one file with a dataset onto a group in a second file and access the dataset via the second file.

```
plist = 'H5P_DEFAULT';
fid2 = H5F.create('file2.h5','H5F_ACC_TRUNC',plist,plist);
gid2 = H5G.create(fid2,'g2',plist,plist,plist);
fid1 = H5F.create('file1.h5','H5F_ACC_TRUNC','H5P_DEFAULT','H5P_DEFAULT')
space_id = H5S.create('H5S_SCALAR');
dset_id = H5D.create(fid1,'DS1','H5T_NATIVE_DOUBLE',space_id,plist);
H5S.close(space_id);
H5D.close(dset_id);
H5F.mount(fid2,'g2',fid1,plist);
dset_id1 = H5D.open(fid1,'/g2/DS1',plist);
H5D.close(dset_id1);
H5F.unmount(fid1,'g2');
H5G.close(gid2);
H5F.close(fid1);
H5F.close(fid2);
```

See Also

`H5F.unmount`

Purpose

Open HDF5 file

Syntax

```
file_id = H5F.open(filename)
file_id = H5F.open(name,flags,fapl_id)
```

Description

`file_id = H5F.open(filename)` opens the file specified by `filename` for read-only access and returns the file identifier, `file_id`.

`file_id = H5F.open(name,flags,fapl_id)` opens the file specified by `name`, returning the file identifier, `file_id`. `flags` specifies file access flags and can be specified by one of the following strings or their numeric equivalents:

'H5F_ACC_RDWR'	read-write mode
'H5F_ACC_RDONLY'	read-only mode

The file access property list, `fapl_id`, may be specified as `'H5P_DEFAULT'`, in which case the default I/O settings are used.

Examples

Open a file in read-only mode with default file access properties.

```
fid = H5F.open('example.h5');
H5F.close(fid);
```

Open a file in read-write mode.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5')
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR','H5P_DEFAULT');
H5F.close(fid);
```

See Also

`H5F.close` | `H5ML.get_constant_value`

H5F.reopen

Purpose

Reopen HDF5 file

Syntax

```
new_file_id = H5F.reopen(file_id)
```

Description

`new_file_id = H5F.reopen(file_id)` returns a new file identifier for the already open HDF5 file specified by `file_id`.

See Also

`H5F.open`

H5F.set_mdc_config

Purpose

Configure HDF5 file metadata cache

Syntax

`H5F.set_mdc_config(fileId,config)`

Description

`H5F.set_mdc_config(fileId,config)` attempts to configure the file's metadata cache according to the supplied configuration structure. Before using this function, you should retrieve the current configuration using `H5F.get_mdc_config`.

See Also

`H5F.get_mdc_config`

H5F.unmount

Purpose

Unmount file or group from mount point

Syntax

```
H5F.unmount(loc_id,name)
```

Description

`H5F.unmount(loc_id,name)` dissassociates the file or group specified by `loc_id` from the mount point specified by `name`. `loc_id` can be a file or group identifier.

See Also

`H5F.mount`

Purpose

Close group

Syntax

```
H5G.close(group_id)
```

Description

`H5G.close(group_id)` releases resources used by the group specified by `group_id`. `group_id` was returned by either `H5G.create` or `H5G.open`.

See Also

`H5G.create` | `H5G.open`

H5G.create

Purpose

Create group

Syntax

```
group_id = H5G.create(loc_id,name,size_hint)
group_id = H5G.create(loc_id,name,lcpl_id,gcpl_id,gapl_id)
```

Description

`group_id = H5G.create(loc_id,name,size_hint)` creates a new group with the name specified by `name` at the location specified by `loc_id`. `loc_id` can be a file or group identifier. `size_hint` specifies the number of bytes to reserve for the names that will appear in the group. This interface corresponds to the 1.6 version of `H5Gcreate`.

`group_id = H5G.create(loc_id,name,lcpl_id,gcpl_id,gapl_id)` creates a new group with link creation, group creation, and group access property lists `lcpl_id`, `gcpl_id`, and `gapl_id`. This interface corresponds to the 1.8 version of `H5Gcreate`.

Examples

Create an HDF5 file `'myfile.h5'` with a group `'my_group'` with default property list settings.

```
fid = H5F.create('myfile.h5');
plist = 'H5P_DEFAULT';
gid = H5G.create(fid,'my_group',plist,plist,plist);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5G.open` | `H5G.close`

Purpose

Information about group

Syntax

```
info = H5G.get_info(group_id)
```

Description

`info = H5G.get_info(group_id)` retrieves information about the group specified by `group_id`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g2');
info = H5G.get_info(gid);
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5G.open` | `H5G.create`

H5G.open

Purpose

Open specified group

Syntax

```
group_id = H5G.open(loc_id,name)
group_id = H5G.open(loc_id,name,gapl_id)
```

Description

`group_id = H5G.open(loc_id,name)` opens the group specified by `name` at the location specified by `loc_id`. `loc_id` is a file or group identifier. This interface corresponds to the 1.6 version of `H5Gopen`.

`group_id = H5G.open(loc_id,name,gapl_id)` opens the group with an additional group access property list, `gapl_id`. This interface corresponds to the 1.8 version of `H5Gopen`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g2');
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5G.close` | `H5P.create`

H5I.dec_ref

Purpose

Decrement reference count

Syntax

```
ref_count = H5I.dec_ref(obj_id)
```

Description

`ref_count = H5I.dec_ref(obj_id)` decrements the reference count of the object identified by `obj_id` and returns the new count.

See Also

`H5I.get_ref` | `H5I.inc_ref`

H5I.get_file_id

Purpose

File identifier for specified object

Syntax

```
file_id = H5I.get_file_id(obj_id)
```

Description

`file_id = H5I.get_file_id(obj_id)` returns the identifier of the file associated with the object referenced by `obj_id`.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g4');
fid2 = H5I.get_file_id(gid);
name = H5F.get_name(fid2);
fprintf('The filename is %s.\n',name);
H5G.close(gid);
H5F.close(fid);
H5F.close(fid2);
```

Purpose

Name of object

Syntax

```
name = H5I.get_name(obj_id)
```

Description

`name = H5I.get_name(obj_id)` returns the name of the object specified by `obj_id`. If no name is attached to the object, the empty string is returned.

Examples

Display the names of all the objects in the `/g3` group in the example file by alphabetical order.

```
idx_type = 'H5_INDEX_NAME';
order = 'H5_ITER_INC';
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g3');
info = H5G.get_info(gid);
for j = 1:info.nlinks
    obj_id = H5O.open_by_idx(fid,'g3',idx_type,order,j-1,'H5P_DEFAULT
    name = H5I.get_name(obj_id);
    fprintf('Object %d: ''%s''.\n',j-1,name);
    H5O.close(obj_id);
end
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5A.get_name` | `H5F.get_name`

H5I.get_ref

Purpose

Reference count of object

Syntax

```
refcount = H5I.get_ref(obj_id)
```

Description

`refcount = H5I.get_ref(obj_id)` returns the reference count of the object specified by `obj_id`.

See Also

`H5I.dec_ref` | `H5I.inc_ref`

Purpose

Type of object

Syntax

```
obj_type = H5I.get_type(obj_id)
```

Description

`obj_type = H5I.get_type(obj_id)` returns the type of the object specified by `obj_id`. `obj_type` corresponds to one of the following enumerated values.

```
H5I_FILE
H5I_GROUP
H5I_DATATYPE
H5I_DATASPACE
H5I_DATASET
H5I_ATTR
H5I_BADID
```

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g3');
dset_id = H5D.open(fid,'/g4/world');
[~,objs] = H5F.get_obj_ids(fid,'H5F_OBJ_ALL',3);
for j = 1:numel(objs)
    name = H5I.get_name(objs(j));
    fprintf('object ''%s'':  ==> ',name);
    type = H5I.get_type(objs(j));
    switch(type)
        case H5ML.get_constant_value('H5I_FILE')
            fprintf('FILE identifier.\n');
        case H5ML.get_constant_value('H5I_GROUP')
            fprintf('GROUP identifier.\n');
        case H5ML.get_constant_value('H5I_DATASET')
            fprintf('DATASET identifier.\n');
        otherwise
            fprintf('unknown identifier type.\n');
    end
```

```
end
H5G.close(gid);
H5F.close(fid);
```

See Also H5ML.get_constant_value

Purpose Increment reference count of specified object

Syntax

```
ref_count = H5I.inc_ref(obj_id)
```

Description `ref_count = H5I.inc_ref(obj_id)` increments the reference count of the specified by `obj_id` and returns the new count.

See Also `H5I.dec_ref` | `H5I.get_ref`

H5I.is_valid

Purpose

Determine if specified identifier is valid

Syntax

```
tf = H5I.is_valid(obj_id)
```

Description

`tf = H5I.is_valid(obj_id)` determines whether the identifier `obj_id` is valid.

Examples

```
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.close(fapl);
if H5I.is_valid(fapl);
    fprintf('File access property list is valid.\n');
else
    fprintf('File access property list is not valid.\n');
end
```

Purpose

Copy link from source location to destination location

Syntax

```
H5L.copy(src_loc_id,src_name,dest_loc_id,dest_name,lcpl_id,
   lapl_id)
```

Description

`H5L.copy(src_loc_id,src_name,dest_loc_id,dest_name,lcpl_id,lapl_id)` copies the link specified by `src_name` from the file or group specified by `src_loc_id` to the destination `dest_loc_id`. The new copy of the link is created with the name `dest_name`.

`dest_loc_id` must refer to either the current file or a group in the current file. If `dest_loc_id` is the file identifier, the copy is placed in the file's root group.

The new link is created with the creation and access property lists specified by `lcpl_id` and `lapl_id`.

Examples

```
plist_id = 'H5P_DEFAULT';
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',plist_id,plist_id);
g1 = H5G.create(fid,'g1',plist_id);
g2 = H5G.create(fid,'g2',plist_id);
g11 = H5G.create(g1,'g3',plist_id);
H5L.copy(g1,'g3',g2,'g4',plist_id,plist_id);
```

H5L.create_external

Purpose Create soft link to external object

Syntax

```
H5L.create_external(filename,objname,link_loc_id,link_name,
   lcpl_id,lapl_id)
```

Description

`H5L.create_external(filename,objname,link_loc_id,link_name,lcpl_id,lapl_id)` creates a soft link to an object in a different file. `filename` identifies the target file containing the target object. `obj_name` specifies the path to the target object within that file. `obj_name` must start at the target file's root group but is not interpreted until lookup time.

`link_loc_id` and `link_name` specify the location and name, respectively, of the new link. `link_name` is interpreted relative to `link_loc_id`.

`lcpl_id` and `lapl_id` are the link creation and access property lists associated with the new link.

Examples

```
plist_id = 'H5P_DEFAULT';
fid1 = H5F.create('myfile1.h5');
g1 = H5G.create(fid1,'g1',plist_id,plist_id,plist_id);
H5G.close(g1);
H5F.close(fid1);
fid2 = H5F.create('myfile2.h5');
H5L.create_external('myfile1.h5','g1',fid2,'g2',plist_id,plist_id);
```

H5L.create_hard

Purpose

Create hard link

Syntax

```
H5L.create_hard(obj_loc_id,obj_name,link_loc_id,link_name,
   lcpl_id,lapl_id)
```

Description

`H5L.create_hard(obj_loc_id,obj_name,link_loc_id,link_name,lcpl_id,lapl` creates a new hard link to a pre-existing object in an HDF5 file. The new link may be one of many that point to that object. `obj_loc_id` and `obj_name` specify the location and name, respectively, of the target object, i.e., the object to which the new hard link points.

`link_loc_id` and `link_name` specify the location and name, respectively, of the new link. `link_name` is interpreted relative to `link_loc_id`.

`lcpl_id` and `lapl_id` are the link creation and access property lists associated with the new link.

Examples

```
fid = H5F.create('myfile.h5');
gid1 = H5G.create(fid,'/g1',0);
gid2 = H5G.create(gid1,'g2',0);
gid3 = H5G.create(gid2,'g3',0);
lcpl = 'H5P_DEFAULT';
lapl = 'H5P_DEFAULT';
H5L.create_hard(gid2,'g3',gid1,'g4',lcpl,lapl);
H5G.close(gid3);
H5G.close(gid2);
H5G.close(gid1);
H5F.close(fid);
```

See Also

`H5L.create_soft`

H5L.create_soft

Purpose

Create soft link

Syntax

```
H5L.create_soft(target_path,link_loc_id,link_name,lcpl_id,
   lapl_id)
```

Description

`H5L.create_soft(target_path,link_loc_id,link_name,lcpl_id,lapl_id)` creates a new soft link to an object in an HDF5 file. The new link may be one of many that point to that object. `target_path` specifies the path to the target object, i.e., the object that the new soft link points to. `target_path` can be anything and is interpreted at lookup time. This `target_path` may be absolute in the file or relative to `link_loc_id`.

`link_loc_id` and `link_name` specify the location and name, respectively, of the new link. `link_name` is interpreted relative to `link_loc_id`.

`lcpl_id` and `lapl_id` are the link creation and access property lists associated with the new link.

Examples

```
plist_id = 'H5P_DEFAULT';
fid = H5F.create('myfile.h5');
gid1 = H5G.create(fid,'/g1',0);
gid3 = H5G.create(gid1,'g3',0);
gid2 = H5G.create(fid,'/g2',0);
lcpl = 'H5P_DEFAULT';
lapl = 'H5P_DEFAULT';
H5L.create_soft('/g1/g3',gid2,'g4',lcpl,lapl);
H5G.close(gid3);
H5G.close(gid2);
H5G.close(gid1);
H5F.close(fid);
```

See Also

`H5L.create_hard`

Purpose Remove link

Syntax H5L.delete(loc_id,name,lapl_id)

Description H5L.delete(loc_id,name,lapl_id) removes the link specified by name from the location loc_id. lapl_id is a link access property list identifier.

Examples Remove the only link to the '/g3' group in example.h5.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5')
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR','H5P_DEFAULT');
H5L.delete(fid,'g3','H5P_DEFAULT');
H5F.close(fid);
```

See Also H5L.move

H5L.exists

Purpose Determine if link exists

Syntax

```
bool = H5L.exists(loc_id,name,lapl_id)
```

Description

`bool = H5L.exists(loc_id,name,lapl_id)` checks if a link specified by the pairing of an object id and name exists within a group. `lapl_id` is a link access property list identifier.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g1/g1.2/g1.2.1');
if H5L.exists(gid,'slink','H5P_DEFAULT')
    fprintf('link exists\n');
else
    fprintf('link does not exist\n');
end
```

Purpose Information about link

Syntax

```
linkStruct = H5L.get_info(location_id,link_name,lapl_id)
```

Description

`linkStruct = H5L.get_info(location_id,link_name,lapl_id)` returns information about a link.

A file or group identifier, `location_id`, specifies the location of the link. `link_name`, interpreted relative to `link_id`, specifies the link being queried.

Examples

```
fid = H5F.open('example.h5');
info = H5L.get_info(fid,'g3','H5P_DEFAULT');
H5F.close(fid);
```

H5L.get_name_by_idx

Purpose

Information about link specified by index

Syntax

```
name = H5L.get_name_by_idx(loc_id,group_name,idx_type,order,n,
   lapl_id)
```

Description

`name =`
`H5L.get_name_by_idx(loc_id,group_name,idx_type,order,n,lapl_id)` retrieves information about a link at index `n`, present in group `group_name`, at location `loc_id`. The `lapl_id` input specifies the link access property list for querying the group.

`idx_type` is the type of index and valid values include the following.

'H5_INDEX_NAME'	Alpha-numeric index on name
'H5_INDEX_CRT_ORDER'	Index on creation order

`order` specifies the index traversal order. Valid values include the following.

'H5_ITER_INC'	Iteration from beginning to end
'H5_ITER_DEC'	Iteration from end to beginning
'H5_ITER_NATIVE'	Iteration in the fastest available order

Examples

```
fid = H5F.open('example.h5');
idx_type = 'H5_INDEX_NAME';
order = 'H5_ITER_DEC';
lapl_id = 'H5P_DEFAULT';
name = H5L.get_name_by_idx(fid,'g3',idx_type,order,0,lapl_id);
H5F.close(fid);
```

Purpose

Value of symbolic link

Syntax

```
linkval = H5L.get_val(link_loc_id,link_name,lapl_id)
```

Description

`linkval = H5L.get_val(link_loc_id,link_name,lapl_id)` returns the value of a symbolic link.

`link_loc_id` is a file or group identifier. `link_name` identifies a symbolic link and is defined relative to `link_loc_id`. Symbolic links include soft and external links and some user-defined links.

In the case of soft links, `linkval` is a cell array containing the path to which the link points.

In the case of external links, `linkval` is a cell array consisting of the name of the target file and the object name.

This function corresponds to the `H5L.get_val` and `H5Lunpack_elink_val` functions in the HDF5 1.8 C API.

Examples

```
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/g1/g1.2/g1.2.1');
linkval = H5L.get_val(gid,'slink','H5P_DEFAULT');
H5G.close(gid);
H5F.close(fid);
```

H5L.iterate

Purpose

Iterate over links

Syntax

```
[status,idx_out,opdata_out] = H5L.iterate(group_id,index_type,
   order,idx_in,iter_func,opdata_in)
```

Description

`[status,idx_out,opdata_out] = H5L.iterate(group_id,index_type,order,idx_in,iter_func,opdata_in)` iterates through the links in a group, specified by `group_id`, to perform a common function whose function handle is `iter_func`. `H5L.iterate` does not recursively follow links into subgroups of the specified group.

`index_type` and `order` establish the iteration. `index_type` specifies the index to be used. If the links have not been indexed by the index type, they will first be sorted by that index then the iteration will begin. If the links have been so indexed, the sorting step will be unnecessary, so the iteration may begin more quickly. Valid values include the following:

'H5_INDEX_NAME'	Alpha-numeric index on name
'H5_INDEX_CRT_ORDER'	Index on creation order

`order` specifies the order in which objects are to be inspected along the index specified in `index_type`. Valid values include the following:

'H5_ITER_INC'	Increasing order
'H5_ITER_DEC'	Decreasing order
'H5_ITER_NATIVE'	Fastest available order

`idx_in` specifies the starting point of the iteration. `idx_out` returns the point at which iteration was stopped. This allows an interrupted iteration to be resumed.

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] =
iter_func(group_id,name,opdata_in)
```

opdata_in is a user-defined value or structure and is passed to the first step of the iteration in the iter_func opdata_in parameter. The opdata_out of an iteration step forms the opdata_in for the next iteration step. The final opdata_out at the end of the iteration is then returned to the caller as opdata_out.

status value returned by iter_func is interpreted as follows:

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

H5L.iterate_by_name

Purpose

Iterate through links in group specified by name

Syntax

```
[status,idx_out,opdata_out] = H5L.iterate_by_name(loc_id,
   group_name,index_type,order,idx_in,iter_func,opdata_in,
   lapl_id)
```

Description

`[status,idx_out,opdata_out] = H5L.iterate_by_name(loc_id,group_name,index_type,order,idx_in,iter_func,o`
iterates through the links in a group to perform a common function whose function handle is `iter_func`. The starting point of the iteration is pairing of a specified by the location id and a relative group name. `H5L.iterate_by_name` does not recursively follow links into subgroups of the specified group. A link access property list, `lapl_id`, may affect the outcome depending upon the type of link being traversed.

`index_type` and `order` establish the iteration. `index_type` specifies the index to be used. If the links have not been indexed by the index type, they will first be sorted by that index then the iteration will begin. If the links have been so indexed, the sorting step will be unnecessary, so the iteration may begin more quickly. Valid values include the following:

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

`order` specifies the order in which objects are to be inspected along the index specified in `index_type`. Valid values include the following:

`'H5_ITER_INC'`	Increasing order
`'H5_ITER_DEC'`	Decreasing order
`'H5_ITER_NATIVE'`	Fastest available order

`idx_in` specifies the starting point of the iteration. `idx_out` returns the point at which iteration was stopped. This allows an interrupted iteration to be resumed.

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] =
iter_func(group_id,name,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final opdata_out at the end of the iteration is then returned to the caller as `opdata_out`.

`status` value returned by `iter_func` is interpreted as follows:

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

H5L.move

Purpose

Rename link

Syntax

```
H5L.move(src_loc_id,src_name,dest_loc_id,dest_name,lcpl_id,
   lapl_id)
```

Description

`H5L.move(src_loc_id,src_name,dest_loc_id,dest_name,lcpl_id,lapl_id)` renames a link within an HDF5 file. The original link, `src_name`, is removed from the group graph and the new link, `dest_name`, is inserted; this change is accomplished as an atomic operation.

`src_loc_id` and `src_name` identify the existing link. `src_loc_id` is either a file or group identifier; `src_name` is the path to the link and is interpreted relative to `src_loc_id`.

`dest_loc_id` and `dest_name` identify the new link. `dest_loc_id` is either a file or group identifier; `dest_name` is the path to the link and is interpreted relative to `dest_loc_id`.

`lcpl_id` and `lapl_id` are the link creation and link access property lists, respectively, associated with the new link, `dest_name`.

Examples

Rename the `'/g2'` group to `'/g2/g3'`.

```
srcFile = fullfile(matlabroot,'toolbox','matlab','demos','example.h5');
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
fid = H5F.open('myfile.h5','H5F_ACC_RDWR','H5P_DEFAULT');
g2id = H5G.open(fid,'g2');
H5L.move(fid,'g3',g2id,'g3','H5P_DEFAULT','H5P_DEFAULT');
H5G.close(g2id);
H5F.close(fid);
```

See Also

`H5L.delete`

Purpose

Recursively iterate through links in group specified by group identifier

Syntax

```
[status opdata_out] = H5L.visit(group_id,index_type,order,
   iter_func,opdata_in)
```

Description

`[status opdata_out] = H5L.visit(group_id,index_type,order,iter_func,opdata_in)` recursively iterates through all links in and below a group, specified by `group_id`, to perform a common function whose function handle is `iter_func`.

`index_type` and order establish the iteration. `index_type` specifies the index to be used. If the links have not been indexed by the index type, they are first sorted by that index, and then the iteration will begin. If the links have been so indexed, the sorting step is unnecessary, so the iteration can begin more quickly. Valid values include the following.

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

Note that the index type passed in `index_type` is a best effort setting. If the application passes in a value indicating iteration in creation order and a group is encountered that was not tracked in creation order, that group will be iterated over in alpha-numeric order by name, or name order. (Name order is the native order used by the HDF5 Library and is always available.)

`order` specifies the order in which objects are to be inspected along the index specified in `index_type`. Valid values include the following.

`'H5_ITER_INC'`	Increasing order
`'H5_ITER_DEC'`	Decreasing order
`'H5_ITER_NATIVE'`	Fastest available order

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] =
iter_func(group_id,name,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final `opdata_out` at the end of the iteration is then returned to the caller as `opdata_out`.

`status` value returned by `iter_func` is interpreted as follows.

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

Purpose

Recursively iterate through links in group specified by location and group name

Syntax

```
[status,opdata_out] = H5L.visit_by_name(loc_id,group_name,
   index_type,order,iter_func,opdata_in,lapl_id)
```

Description

`[status,opdata_out] = H5L.visit_by_name(loc_id,group_name,index_type,order,iter_func,opdata_` recursively iterates though all links in and below a group to perform a common function whose function handle is `iter_func`. The starting point of the iteration is specified by the pairing of a location id and a relative group name. A link access property list, `lapl_id`, may affect the outcome depending upon the type of link being traversed.

`index_type` and `order` establish the iteration. `index_type` specifies the index to be used. If the links have not been indexed by the index type, they are first sorted by that index, and then the iteration will begin. If the links have been so indexed, the sorting step is unnecessary, so the iteration can begin more quickly. Valid values include the following.

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

Note that the index type passed in `index_type` is a best effort setting. If the application passes in a value indicating iteration in creation order and a group is encountered that was not tracked in creation order, that group will be iterated over in alpha-numeric order by name, or name order. (Name order is the native order used by the HDF5 Library and is always available.)

`order` specifies the order in which objects are to be inspected along the index specified in `index_type`. Valid values include the following.

`'H5_ITER_INC'`	Increasing order
`'H5_ITER_DEC'`	Decreasing order
`'H5_ITER_NATIVE'`	Fastest available order

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] =
iter_func(group_id,name,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final `opdata_out` at the end of the iteration is then returned to the caller as `opdata_out`.

`status` value returned by `iter_func` is interpreted as follows.

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

H5ML.compare_values

Purpose

Numerically compare two HDF5 values

Syntax

```
bEqual = H5ML.compare_values(value1,value2)
```

Description

`bEqual = H5ML.compare_values(value1,value2)` compares two values, where either or both values may be represented as a string. The values are compared numerically.

Function parameters:

`bEqual`	A logical value indicating whether the two values are equal
`value1`	The first value to be compared
`value2`	The second value to be compared

Examples

```
val = H5ML.get_constant_value('H5T_NATIVE_INT');
H5ML.compare_values(val,'H5T_NATIVE_INT')
```

H5ML.get_constant_names

Purpose

Constants known by HDF5 library

Syntax

```
names = H5ML.get_constant_names()
```

Description

`names = H5ML.get_constant_names()` returns a list of known library constants, definitions, and enumerations. When these strings are supplied as actual parameters to HDF5 functions, they are automatically be converted to the appropriate numeric value.

Function parameters.

`names`	An alphabetized cell array of names

Purpose Value corresponding to a string

Syntax

```
value = H5ML.get_constant_value(constant)
```

Description `value = H5ML.get_constant_value(constant)` returns the value corresponding to a given sting. The string should correspond to an enumeration (for example, `'H5_ENUM_T'`) or a predefined identifier (for example, `'H5T_NATIVE_INT'`). Since the value corresponding to a given string is not guaranteed to remain the same, it is almost always preferable to use the `H5ML.compare_values()` function instead.

Function parameters:

`value`	The value corresponding to the supplied string
`constant`	A string that corresponds to a HDF5 enumeration or defined value.

Examples

```
a = H5ML.get_constant_value('H5T_NATIVE_INT');
```

H5ML.get_function_names

Purpose

Functions provided by HDF5 library

Syntax

```
names = H5ML.get_function_names()
```

Description

`names = H5ML.get_function_names()` returns a list of supported library functions.

Function parameters:

`names`	An alphabetized cell array of names

H5ML.get_mem_datatype

Purpose

Data type for dataset ID

Syntax

```
DTYPE_ID = H5ML.get_mem_datatype(LOCATION_ID)
```

Description

`DTYPE_ID = H5ML.get_mem_datatype(LOCATION_ID)` returns the ID of an HDF5 memory datatype for the dataset or attribute identified by `LOCATION_ID`. This HDF5 memory datatype is the default used by `H5D.read` or `H5D.write` when you specify `'H5ML_DEFAULT'` as a value of the memory data type parameter.

The identifier returned by `H5ML.get_mem_datatype` should eventually be closed by calling `H5T.close` to release resources.

Examples

```
file_id = H5F.open('example.h5','H5F_ACC_RDONLY','H5P_DEFAULT');
dset_id = H5D.open(file_id,'/g1/g1.1/dset1.1.1');
datatype_id = H5ML.get_mem_datatype(dset_id)
H5T.close(datatype_id);
H5D.close(dset_id);
H5F.close(file_id);
```

H5ML.hoffset

Purpose

Determine the offset of a field within a structure

Note `H5ML.hoffset` is not recommended. Use `H5T` instead.

Syntax

Description

This function is used to determine the offset (in bytes) of a structure, `H5T.insert(file_type,'a', offset(1), dtype(1));`, within a field. It is used when constructing an HDF5 COMPOUND type. It is designed to correspond to the HDF5 HOFFSET macro. For more details about the operation of the HOFFSET macro, please consult the HDF5 documentation.

Function parameters:

`offset`	The byte offset of the field within the structure.
`structure`	The structure which contains the specified fieldname.
`fieldname`	The field for which the offset is determined.

Examples

This function is deprecated. It can only be used in workflows that do not include a field that is itself an HDF5 COMPOUND or of variable length. To handle these cases, the offsets should be computed directly. For example, in the case above, a file dataspace for such a compound could be created with:

```
dtype(1) = H5T.copy('H5T_NATIVE_INT');
dtype(2) = H5T.copy('H5T_NATIVE_DOUBLE');
dtype(3) = H5T.copy('H5T_NATIVE_FLOAT');

for j = 1:3, sz(j,1) = H5T.get_size(dtype(j)); end
```

```
% The first offset would always be zero and the size of the last
% field does not matter.
offset(1) = 0;
offset(2:3) = cumsum(sz(1:2));

file_type = H5T.create('H5T_COMPOUND',sum(sz));

H5T.insert(file_type,'a', offset(1), dtype(1));
H5T.insert(file_type,'b', offset(2), dtype(2));
H5T.insert(file_type,'c', offset(3), dtype(3));
```

See Also

H5T.get_size

H5ML.sizeof

Purpose

Return the size (in bytes) of a variable as stored on disk

Note `H5ML.sizeof` is not recommended. Use `H5T` instead.

Syntax

Description

This function is used to determine the size (in bytes) of a structure or other (simple) variable. It is designed to correspond to the C sizeof() operator as it is used during the creation of HDF5 datatypes, especially the HDF5 COMPOUND type.

Function parameters:

`size`	The size (in bytes) of the variable as it would be stored on disk.
`arg`	The variable for which the size is being sought.

Examples

This function is deprecated. It can only be used in workflows that do not include a field that is itself an HDF5 COMPOUND or of variable length. To handle these cases, the offsets should be computed directly. For example, in the case above, a file dataspace for such a compound could be created with:

```
dtype(1) = H5T.copy('H5T_NATIVE_INT');
dtype(2) = H5T.copy('H5T_NATIVE_DOUBLE');
dtype(3) = H5T.copy('H5T_NATIVE_FLOAT');

for j = 1:3, sz(j,1) = H5T.get_size(dtype(j)); end

% The first offset would always be zero and the size of the last
% field does not matter.
offset(1) = 0;
offset(2:3) = cumsum(sz(1:2));
```

```
file_type = H5T.create('H5T_COMPOUND',sum(sz));

H5T.insert(file_type,'a', offset(1), dtype(1));
H5T.insert(file_type,'b', offset(2), dtype(2));
H5T.insert(file_type,'c', offset(3), dtype(3));
```

See Also

`H5T.get_size`

H5O.close

Purpose Close object

Syntax `H5O.close(obj_id)`

Description `H5O.close(obj_id)` closes the object specified by `obj_id`. `obj_id` cannot be a dataspace, attribute, property list, or file.

Purpose

Copy object from source location to destination location

Syntax

```
H5O.copy(src_loc_id,src_name,dst_loc_id,dst_name,ocpypl_id,
   lcpl_id)
```

Description

`H5O.copy(src_loc_id,src_name,dst_loc_id,dst_name,ocpypl_id,lcpl_id)` copies the group, dataset or named datatype specified by `src_name` from the file or group specified by `src_loc_id` to the destination location `dst_loc_id`.

The destination location, as specified in `dst_loc_id`, may be a group in the current file or a location in a different file. If `dst_loc_id` is a file identifier, the copy will be placed in that file's root group.

The new copy will be created with the name `dst_name`. `dst_name` must not pre-exist in the destination location. If `dst_name` already exists at the location `dst_loc_id`, the operation will fail.

The new copy of the object is created with the object creation property and link creation property lists `ocpypl_id` and `lcpl_id`, respectively.

Examples

Copy the group `'/g3'` and all its datasets to a new group `'/g3.5'`.

```
srcFile = [matlabroot '/toolbox/matlab/demos/example.h5'];
copyfile(srcFile,'myfile.h5');
fileattrib('myfile.h5','+w');
ocpl = H5P.create('H5P_OBJECT_COPY');
lcpl = H5P.create('H5P_LINK_CREATE');
H5P.set_create_intermediate_group(lcpl,true);
fid = H5F.open('myfile.h5','H5F_ACC_RDWR','H5P_DEFAULT');
gid = H5G.open(fid,'/');
H5O.copy(gid,'g3',gid,'g3.5',ocpl,lcpl);
H5G.close(gid);
H5P.close(ocpl);
H5P.close(lcpl);
H5F.close(fid);
```

H5O.get_comment

Purpose

Get comment for object specified by object identifier

Syntax

```
comment = H5O.get_comment(obj_id)
```

Description

`comment = H5O.get_comment(obj_id)` retrieves the comment for the object specified by `obj_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'g4/world');
comment = H5O.get_comment(dset_id);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5O.get_comment_by_name` | `H5O.set_comment` | `H5O.set_comment_by_name`

H5O.get_comment_by_name

Purpose

Get comment for object specified by location and object name

Syntax

```
comment = H5O.get_comment_by_name(loc_id,name,lapl_id)
```

Description

`comment = H5O.get_comment_by_name(loc_id,name,lapl_id)` retrieves a comment where a location id and name together specify the object. A link access property list can affect the outcome if a link is traversed to access the object.

Examples

```
fid = H5F.open('example.h5','H5F_ACC_RDONLY','H5P_DEFAULT');
comment = H5O.get_comment_by_name(fid,'g4/world','H5P_DEFAULT');
H5F.close(fid);
```

See Also

`H5O.get_comment` | `H5O.set_comment` | `H5O.set_comment_by_name`

H5O.get_info

Purpose

Object metadata

Syntax

```
info = H5O.get_info(obj_id)
```

Description

`info = H5O.get_info(obj_id)` retrieves the metadata for an object specified by `obj_id`. For details about the object metadata, please refer to the HDF5 documentation.

Examples

Determine the number of attributes for a dataset.

```
fid = H5F.open('example.h5','H5F_ACC_RDONLY','H5P_DEFAULT');
dsetId = H5D.open(fid,'/g1/g1.1/dset1.1.1');
info = H5O.get_info(dsetId);
info.num_attrs
```

Determine the type of objects in the root group.

```
plist = 'H5P_DEFAULT';
fid = H5F.open('example.h5');
gid = H5G.open(fid,'/');
root_info = H5G.get_info(gid);
idx_type = 'H5_INDEX_NAME';
order = 'H5_ITER_DEC';
for j = 0:root_info.nlinks-1
   obj_id = H5O.open_by_idx(fid,'/',idx_type,order,j,plist);
   obj_info = H5O.get_info(obj_id);
   switch(obj_info.type)
       case H5ML.get_constant_value('H5G_LINK')
           fprintf('Object #%d is a link.\n',j);
       case H5ML.get_constant_value('H5G_GROUP')
           fprintf('Object #%d is a group.\n',j);
       case H5ML.get_constant_value('H5G_DATASET')
           fprintf('Object #%d is a dataset.\n',j);
       case H5ML.get_constant_value('H5G_TYPE')
           fprintf('Object #%d is a named datatype.\n',j);
```

```
   end
   H5O.close(obj_id);
end
H5G.close(gid);
H5F.close(fid);
```

See Also

H5F.open | H5G.open | H5D.open | H5T.open

H5O.link

Purpose

Create hard link to specified object

Syntax

```
H5O.link(obj_id,new_loc_id,new_link_name,lcpl_id,lapl_id)
```

Description

`H5O.link(obj_id,new_loc_id,new_link_name,lcpl_id,lapl_id)` creates a hard link to an object, where `new_loc_id` and `new_link_name` specify the location. `lcpl_id` and `lapl_id` are the link creation and access property lists associated with the new link.

`H5O.link` is designed to add additional structure to an existing file so that, for example, an object can be shared among multiple groups.

Examples

Create a hard link from group `'/g2'` to the dataset `'/g1/ds1'`.

```
plist_id = 'H5P_DEFAULT';
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',plist_id,plist_id);
gid1 = H5G.create(fid,'/g1',plist_id);
space_id = H5S.create_simple(1,10,[]);
ds1 = H5D.create(gid1,'ds1','H5T_NATIVE_INT',space_id,plist_id);
gid2 = H5G.create(fid,'/g2',plist_id);
H5O.link(ds1,gid2,'ds2',plist_id,plist_id);
H5D.close(ds1);
H5S.close(space_id);
H5G.close(gid2); H5G.close(gid1);
H5F.close(fid);
```

See Also

`H5L.create_hard` | `H5L.create_soft`

Purpose

Open specified object

Syntax

```
obj_id = H5O.open(loc_id,relname,lapl_id)
```

Description

`obj_id = H5O.open(loc_id,relname,lapl_id)` opens an object specified by location identifier and relative path name. `lapl_id` is the link access property list associated with the link pointing to the object. If default link access properties are appropriate, this can be passed in as `'H5P_DEFAULT'`.

Examples

```
fid = H5F.open('example.h5');
obj_id = H5O.open(fid,'g3','H5P_DEFAULT');
H5O.close(obj_id);
H5F.close(fid);
```

See Also

`H5O.open_by_idx` | `H5O.close`

H5O.open_by_idx

Purpose

Open object specified by index

Syntax

```
obj_id = H5O.open_by_idx(loc_id, group_name, idx_type, order,
   n, lapl_id)
```

Description

`obj_id = H5O.open_by_idx(loc_id, group_name, idx_type, order, n, lapl_id)` opens the `n`-th object in the group specified by `loc_id` and `group_name`. `loc_id` specifies a file or group. `group_name` specifies the group relative to `loc_id` in which the object can be found.

Two parameters are used to establish the iteration: `index_type` and `order`. `index_type` specifies the type of index by which objects are ordered. Valid values include the following:

'H5_INDEX_NAME'	Alpha-numeric index on name
'H5_INDEX_CRT_ORDER'	Index on creation order

`order` specifies the order in which the links are to be referenced for the purposes of this function. Valid values include the following:

'H5_ITER_INC'	Increasing order
'H5_ITER_DEC'	Decreasing order
'H5_ITER_NATIVE'	Fastest available order

`n` specifies the zero-based position of the object within the index. `lapl_id` specifies the link access property list to be used in accessing the object.

Examples

```
fid = H5F.open('example.h5');
idx_type = 'H5_INDEX_NAME';
order = 'H5_ITER_DEC';
obj_id = H5O.open_by_idx(fid,'g3',idx_type,order,0,'H5P_DEFAULT');
H5O.close(obj_id);
H5F.close(fid);
```

See Also

H5O.open | H5O.close

H5O.set_comment

Purpose

Set comment for object specified by object identifier

Syntax

```
H5O.set_comment(obj_id,comment)
```

Description

`H5O.set_comment(obj_id,comment)` sets a comment for the object specified by `obj_id`.

Examples

```
plist = 'H5P_DEFAULT';
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',plist,plist);
gid = H5G.create(fid,'/g1',plist);
H5O.set_comment(gid,'This is a group comment.');
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5O.get_comment` | `H5O.get_comment_by_name` | `H5O.set_comment_by_name`

H5O.set_comment_by_name

Purpose

Set comment for object specified by location and object name

Syntax

```
H5O.set_comment_by_name(loc_id,rel_name,comment,lapl_id)
```

Description

`H5O.set_comment_by_name(loc_id,rel_name,comment,lapl_id)` sets a comment for an object specified by a location ID and a relative name. `lapl_id` is a link access property list identifier that can affect the outcome if links are traversed.

Examples

```
plist = 'H5P_DEFAULT';
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',plist,plist);
gid = H5G.create(fid,'/g1',plist);
H5O.set_comment_by_name(fid,'g1','This is a group comment.',plist);
H5G.close(gid);
H5F.close(fid);
```

See Also

H5O.get_comment | H5O.get_comment_by_name | H5O.set_comment

Purpose

Visit objects specified by object identifier

Syntax

```
[status,opdata_out] = H5O.visit(obj_id,index_type,order,
   iter_func,opdata_in)
```

Description

`[status,opdata_out] = H5O.visit(obj_id,index_type,order,iter_func,opdata_in)` is a recursive iteration function to visit the object `object_id` and, if `object_id` is a group, all objects in and below it in an HDF5 file. This provides a mechanism for an application to perform a common set of operations across all of those objects or a dynamically selected subset.

If `object_id` is a group identifier, that group serves as the root of a recursive iteration. If `object_id` is a file identifier, that file's root group serves as the root of the recursive iteration. If `object_id` is any other type of object, such as a dataset or named data type, there is no iteration.

Two parameters are used to establish the iteration: `index_type` and `order`. The `index_type` parameter specifies the index used. If the links in a group have not been indexed by the index type, they are first sorted by that index, and then the iteration will begin. If the links have been so indexed, the sorting step is unnecessary, so the iteration can begin more quickly. Valid values include the following.

'H5_INDEX_NAME'	Alpha-numeric index on name
'H5_INDEX_CRT_ORDER'	Index on creation order

Note that the index type passed in `index_type` is a best effort setting. If the application passes in a value indicating iteration in creation order and a group is encountered that was not tracked in creation order, that group will be iterated over in alpha-numeric order by name, or name order. (Name order is the native order used by the HDF5 Library and is always available.) `order` specifies the order in which objects are to be inspected along the index specified in `index_type`. Valid values include the following.

`'H5_ITER_INC'`	Increasing order
`'H5_ITER_DEC'`	Decreasing order
`'H5_ITER_NATIVE'`	Fastest available order

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] =
iter_func(group_id,name,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final `opdata_out` at the end of the iteration is then returned to the caller as `opdata_out`.

`status` value returned by `iter_func` is interpreted as follows.

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

See Also

`H5O.visit_by_name`

Purpose

Visit objects specified by location and object name

Syntax

```
[status,opdata_out] = H5O.visit_by_name(loc_id,obj_name,
   index_type,order,iter_func,opdata_in,lapl_id)
```

Description

`[status,opdata_out] =`
`H5O.visit_by_name(loc_id,obj_name,index_type,order,iter_func,opdata_in` specifies the object by the pairing of the location identifier and object name. `loc_id` specifies a file or an object in a file and `obj_name` specifies an object in the file with either an absolute name or relative to `loc_id`. A link access property list can affect the outcome if links are involved.

Two parameters are used to establish the iteration: `index_type` and `order`. The `index_type` parameter specifies the index to be used. If the links in a group have not been indexed by the index type, they are first sorted by that index, and then the iteration will begin; if the links have been so indexed, the sorting step is unnecessary, so the iteration can begin more quickly. Valid values include the following.

`'H5_INDEX_NAME'`	Alpha-numeric index on name
`'H5_INDEX_CRT_ORDER'`	Index on creation order

Note that the index type passed in `index_type` is a best effort setting. If the application passes in a value indicating iteration in creation order and a group is encountered that was not tracked in creation order, that group will be iterated over in alpha-numeric order by name, or name order. (Name order is the native order used by the HDF5 Library and is always available.) `order` specifies the order in which objects are to be inspected along the index specified in `index_type`. Valid values include the following.

`'H5_ITER_INC'`	Increasing order
`'H5_ITER_DEC'`	Decreasing order
`'H5_ITER_NATIVE'`	Fastest available order

The callback function `iter_func` must have the following signature:

```
function [status,opdata_out] =
iter_func(group_id,name,opdata_in)
```

`opdata_in` is a user-defined value or structure and is passed to the first step of the iteration in the `iter_func` `opdata_in` parameter. The `opdata_out` of an iteration step forms the `opdata_in` for the next iteration step. The final `opdata_out` at the end of the iteration is then returned to the caller as `opdata_out`.

`lapl_id` is a link access property list. When default link access properties are acceptable, `'H5P_DEFAULT'` can be used.

`status` value returned by `iter_func` is interpreted as follows.

zero	Continues with the iteration or returns zero status value to the caller if all members have been processed
positive	Stops the iteration and returns the positive status value to the caller
negative	Stops the iteration and throws an error indicating failure

See Also

`H5O.visit`

Purpose Close property list

Syntax `H5P.close(plist_id)`

Description `H5P.close(plist_id)` terminates access to the property list specified by `plist_id`.

See Also `H5P.create`

H5P.copy

Purpose

Copy of property list

Syntax

```
plist_copy = H5P.copy(plist_id)
```

Description

`plist_copy = H5P.copy(plist_id)` returns a copy of the property list specified by `plist_id`.

Examples

Make a copy of the file creation property list for `'example.h5'`.

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
fcpl2 = H5P.copy(fcpl);
```

Purpose Create new property list

Syntax

```
plist = H5P.create(class_id)
```

Description

`plist = H5P.create(class_id)` creates a new property list as an instance of the property list class specified by `class_id`. The `class_id` input can be one of the following strings or the corresponding constant value.

```
'H5P_ATTRIBUTE_CREATE'
'H5P_DATASET_ACCESS'
'H5P_DATASET_CREATE'
'H5P_DATASET_XFER'
'H5P_DATATYPE_CREATE'
'H5P_DATATYPE_ACCESS'
'H5P_FILE_MOUNT'
'H5P_FILE_CREATE'
'H5P_FILE_ACCESS'
'H5P_GROUP_CREATE'
'H5P_GROUP_ACCESS'
'H5P_LINK_CREATE'
'H5P_LINK_ACCESS'
'H5P_OBJECT_COPY'
'H5P_OBJECT_CREATE'
'H5P_STRING_CREATE'
```

`class_id` can also be an instance of a property list class.

Examples

```
fapl = H5P.create('H5P_FILE_ACCESS');
fid = H5F.open('example.h5','H5F_ACC_RDONLY',fapl);
```

See Also

H5P.close | H5P.get_class | H5ML.get_constant_value

H5P.get_class

Purpose Property list class

Syntax

```
plist_class = H5P.get_class(plist_id)
```

Description

`plist_class = H5P.get_class(plist_id)` returns the property list class for the property list specified by `plist_id`.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
pclass = H5P.get_class(fcpl);
name = H5P.get_class_name(pclass);
```

See Also `H5P.get_class_name`

H5P.close_class

Purpose Close property list class

Syntax `H5P.close_class(class)`

Description `H5P.close_class(class)` closes the property list class specified by `pclass_id`.

H5P.equal

Purpose

Determine equality of property lists

Syntax

```
value = H5P.equal(plist1_id, plist2_id)
```

Description

`value = H5P.equal(plist1_id, plist2_id)` returns a positive number if the two property lists specified are equal, and zero if they are not. A negative value indicates failure.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
fcpl = H5F.get_create_plist(fid);
if H5P.equal(fapl,fcpl)
    fprintf('property lists are equal\n');
else
    fprintf('property lists are not equal\n');
end
```

H5P.exist

Purpose

Determine if specified property exists in property list

Syntax

```
value = H5P.exist(prop_id, name)
```

Description

`value = H5P.exist(prop_id, name)` returns a positive value if the property specified by the text string `name` exists within the property list or class specified by `prop_id`.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
if H5P.exist(fapl,'sieve_buf_size')
    fprintf('sieve buffer size property exists\n');
else
    fprintf('sieve buffer size property does not exist\n');
end
```

Purpose

Value of specified property in property list

Syntax

```
value = H5P.get(plist_id, name)
```

Description

`value = H5P.get(plist_id, name)` retrieves a copy of the value of the property specified by the text string `name` in the property list specified by `plist_id`. The `H5P.get` function returns the property as an array of `uint8` values. You might need to cast the value to an appropriate data type to get a meaningful result.

Examples

```
plist = H5P.create('H5P_FILE_ACCESS');
val = H5P.get(plist, 'rdcc_w0');
rdcc_w0 = typecast(val,'double');
```

It is recommended to use alternative functions like `H5P.get_chunk`, `H5P.get_layout`, `H5P.get_size` etc., where available, to get values for the common property names.

See Also

`H5P.set` | `typecast`

H5P.get_class_name

Purpose

Name of property list class

Syntax

```
classname = H5P.get_class_name(pclass_id)
```

Description

`classname = H5P.get_class_name(pclass_id)` retrieves the name of the generic property list class. `classname` is a text string. If no class is found, the empty string is returned.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
pclass = H5P.get_class(fcpl);
name = H5P.get_class_name(pclass);
```

See Also

`H5P.get_class`

H5P.get_class_parent

Purpose

Identifier for parent class

Syntax

```
pclass_obj_id = H5P.get_class_parent(pclass_id)
```

Description

`pclass_obj_id = H5P.get_class_parent(pclass_id)` returns an identifier to the parent class object of the property class specified by `pclass_id`.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
fcpl_class = H5P.get_class(fcpl);
parent_class = H5P.get_class_parent(fcpl_class);
name = H5P.get_class_name(parent_class);
```

See Also

`H5P.get_class` | `H5P.get_class_name`

H5P.get_nprops

Purpose

Query number of properties in property list or class

Syntax

```
nprops = H5P.get_nprops(id)
```

Description

`nprops = H5P.get_nprops(id)` returns the number of properties in the property list or class specified by `id`.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
nprops = H5P.get_nprops(fcpl);
```

Purpose

Query size of property value in bytes

Syntax

```
sz = H5P.get_size(id,name)
```

Description

`sz = H5P.get_size(id,name)` returns the size, in bytes, of the property specified by the text string `name` in the property list or property class specified by `id`.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
sz = H5P.get_size(fapl,'sieve_buf_size');
```

H5P.isa_class

Purpose

Determine if property list is member of class

Syntax

```
output = H5P.isa_class(plist_id, pclass_id)
```

Description

`output = H5P.isa_class(plist_id, pclass_id)` returns a positive number if the property list specified by `plist_id` is a member of the class specified by `pclass_id`, zero if it is not, and a negative value to indicate an error.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
if H5P.isa_class(fcpl,'H5P_FILE_ACCESS')
    fprintf('fcpl is a file access property list\n');
else
    fprintf('fcpl is not a file access property list\n');
end
```

See Also

`H5P.get_class`

Purpose

Iterate over properties in property list

Syntax

```
[output,idx_out] = H5P.iterate(id,idx_in,iter_func)
```

Description

`[output,idx_out] = H5P.iterate(id,idx_in,iter_func)` executes the operation `iter_func` on each property in the property object specified in `id`. The `id` input can be a property list or a property class. `idx_in` specifies the index of the next property to be processed. `output` is the value returned by the last call to `iter_func`. `idx_out` is the index of the last property processed. `iter_func` is a function handle.

The iterator function must have the following signature:

```
status = iter_func(id,prop_name)
```

`id` still identifies the property object passed into `H5P.iterate`, but `name` identifies the name of the current property.

H5P.set

Purpose

Set property list value

Syntax

```
H5P.set(plist_id,name,value)
```

Description

`H5P.set(plist_id,name,value)` sets the value of the property specified by the text string `name` in the property list specified by `plist_id` to the value specified in `value`. The datatype of `value` must be `uint8`.

Examples

```
plist = H5P.create('H5P_FILE_ACCESS');
H5P.set(plist, 'rdcc_w0', typecast(0.8, 'uint8'));
```

It is recommended to use alternative functions like `H5P.set_chunk`, `H5P.set_layout`, `H5P.set_size`, etc., where available, to set values for the common property names.

See Also

`typecast`

Purpose

B-tree split ratios

Syntax

```
[left,middle,right] = H5P.get_btree_ratios(plist_id)
```

Description

`[left,middle,right] = H5P.get_btree_ratios(plist_id)` returns the B-tree split ratios for the dataset transfer property list specified by `plist_id`. The `left` output specifies the B-tree split ratio for left-most nodes. `right` corresponds to the right-most nodes and lone nodes, and `middle` corresponds to all other nodes.

Examples

```
dxpl = H5P.create('H5P_DATASET_XFER');
[left,middle,right] = H5P.get_btree_ratios(dxpl);
```

See Also

`H5P.set_btree_ratios`

H5P.get_chunk_cache

Purpose

Raw data chunk cache parameters

Syntax

```
[rdcc_nslots,rdcc_nbytes,
   rdcc_w0] = H5P.get_chunk_cache(dapl_id)
```

Description

`[rdcc_nslots,rdcc_nbytes,rdcc_w0] = H5P.get_chunk_cache(dapl_id)` retrieves the number of chunk slots in the raw data chunk cache hash table (`rdcc_nslots`), the maximum possible number of bytes in the raw data chunk cache (`rdcc_nbytes`), and the preemption policy value (`rdcc_w0`) on a dataset access property list.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/vlen3D');
dapl = H5D.get_access_plist(dset_id);
[rrdcc_nslots,rdcc_nbytes,rdcc_w0] = H5P.get_chunk_cache(dapl);
H5P.close(dapl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.set_chunk_cache`

Purpose

Data access property lists for multiple files

Syntax

```
memb_dxpl = H5P.get_dxpl_multi(dxpl_id)
```

Description

`memb_dxpl = H5P.get_dxpl_multi(dxpl_id)` returns an array of data access property lists for the multifile data transfer property list specified by `dxpl_id`.

See Also

`H5P.set_dxpl_multi`

H5P.get_edc_check

Purpose

Determine if error detection is enabled

Syntax

```
check = H5P.get_edc_check(plist_id)
```

Description

`check = H5P.get_edc_check(plist_id)` queries the dataset transfer property list, specified by `plist_id`, to determine whether error detection is enabled for data read operations. Returns either `H5Z_ENABLE_EDC` or `H5Z_DISABLE_EDC`.

Examples

```
dxpl = H5P.create('H5P_DATASET_XFER');
check = H5P.get_edc_check(dxpl);
switch(check)
    case H5ML.get_constant_value('H5Z_ENABLE_EDC')
        fprintf('error detection enabled\n');
    case H5ML.get_constant_value('H5Z_DISABLE_EDC');
        fprintf('error detection disabled\n');
end
```

See Also

`H5P.set_edc_check`

Purpose

Number of I/O vectors

Syntax

```
sz = H5P.get_hyper_vector_size(dxpl_id)
```

Description

`sz = H5P.get_hyper_vector_size(dxpl_id)` returns the number of I/O vectors to be read/written in hyperslab I/O.

Examples

```
dxpl = H5P.create('H5P_DATASET_XFER');
sz = H5P.get_hyper_vector_size(dxpl);
```

See Also

`H5P.set_hyper_vector_size`

H5P.set_btree_ratios

Purpose

Set B-tree split ratios for dataset transfer

Syntax

```
H5P.set_btree_ratios(plist_id,left,middle,right)
```

Description

`H5P.set_btree_ratios(plist_id,left,middle,right)` sets the B-tree split ratios for the dataset transfer property list specified by `plist_id`. The `left` argument specifies the B-tree split ratio for left-most nodes. `right` specifies the B-tree split ratio for right-most nodes and lone nodes. `middle` specifies the B-tree split ratio for all other nodes.

Examples

```
dxpl = H5P.create('H5P_DATASET_XFER');
H5P.set_btree_ratios(dxpl, 0.2, 0.6, 0.95);
```

See Also

`H5P.get_btree_ratios`

Purpose

Set raw data chunk cache parameters

Syntax

```
H5P.set_chunk_cache(dapl_id, rdcc_nslots, rdcc_nbytes,
   rdcc_w0)
```

Description

`H5P.set_chunk_cache(dapl_id, rdcc_nslots, rdcc_nbytes, rdcc_w0)` sets the number of elements (`rdcc_nslots`), the total number of bytes (`rdcc_nbytes`), and the preemption policy value (`rdcc_w0`) in the raw data chunk cache.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/vlen3D');
dapl = H5D.get_access_plist(dset_id);
H5P.set_chunk_cache(dapl,500,1e6,0.7);
H5P.close(dapl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.get_chunk_cache`

H5P.set_dxpl_multi

Purpose

Set data transfer property list for multifile driver

Syntax

```
H5P.set_dxpl_multi(dxpl_id, memb_dxpl)
```

Description

`H5P.set_dxpl_multi(dxpl_id, memb_dxpl)` sets the data transfer property list `dxpl_id` to use the multifile driver. `memb_dxpl` is an array of data access property lists.

See Also

`H5P.get_dxpl_multi`

H5P.set_edc_check

Purpose Enable error detection for dataset transfer

Syntax `H5P.set_edc_check(plist_id,check)`

Description `H5P.set_edc_check(plist_id,check)` sets the dataset transfer property list specified by `plist_id` to enable or disable error detection when reading data. `check` can have the value `H5Z_ENABLE_EDC` or `H5Z_DISABLE_EDC`.

Examples Disable error detection for a default dataset transfer property list.

```
dxpl = H5P.create('H5P_DATASET_XFER');
H5P.set_edc_check(dxpl,'H5Z_DISABLE_EDC');
```

See Also `H5P.get_edc_check`

H5P.set_hyper_vector_size

Purpose

Set number of I/O vectors for hyperslab I/O

Syntax

```
H5P.set_hyper_vector_size(dxpl_id,size)
```

Description

`H5P.set_hyper_vector_size(dxpl_id,size)` sets the number of I/O vectors to be accumulated in memory before being issued to the lower levels of the HDF5 library for reading or writing the actual data. `dxpl_id` is a dataset transfer property list identifier. `size` specifies the number of I/O vectors to accumulate in memory for I/O operations.

Examples

```
dxpl = H5P.create('H5P_DATASET_XFER');
H5P.set_hyper_vector_size(dxpl,2048);
```

See Also

`H5P.get_hyper_vector_size`

H5P.all_filters_avail

Purpose

Determine availability of all filters

Syntax

```
value = H5P.all_filters_avail(dcpl_id)
```

Description

`value = H5P.all_filters_avail(dcpl_id)` returns a positive value if all of the filters set in the dataset creation property list `dcpl_id` are currently available, and zero if they are not. A negative value indicates failure.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
dcpl = H5D.get_create_plist(dset_id);
if H5P.all_filters_avail(dcpl)
    fprintf('all filters available\n');
else
    fprintf('all filters not available\n');
end
H5P.close(dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.set_filter`

H5P.fill_value_defined

Purpose Determine if fill value is defined

Syntax

```
fvstatus = H5P.fill_value_defined(plist_id)
```

Description

`fvstatus = H5P.fill_value_defined(plist_id)` determines whether a fill value is defined in the dataset creation property list `plist_id`. The `fvstatus` output can have any of the following values: `H5D_FILL_VALUE_UNDEFINED`, `H5D_FILL_VALUE_DEFAULT`, or `H5D_FILL_VALUE_USER_DEFINED`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
dcpl = H5D.get_create_plist(dset_id);
fvstatus = H5P.fill_value_defined(dcpl);
switch(fvstatus)
    case H5ML.get_constant_value('H5D_FILL_VALUE_UNDEFINED')
        fprintf('fill value undefined\n');
    case H5ML.get_constant_value('H5D_FILL_VALUE_DEFAULT')
        fprintf('fill value set to default\n');
    case H5ML.get_constant_value('H5D_FILL_VALUE_USER_DEFINED')
        fprintf('fill value is user defined\n');
end
```

See Also

`H5P.get_fill_value` | `H5P.set_fill_value`

Purpose Return timing of storage space allocation

Syntax

```
alloc_time = H5P.get_alloc_time(plist_id)
```

Description `alloc_time = H5P.get_alloc_time(plist_id)` retrieves the timing for storage space allocation from the dataset creation property list specified by `plist_id`. The `alloc_time` output can have any of the following values: `H5D_ALLOC_TIME_DEFAULT`, `H5D_ALLOC_TIME_EARLY`, `H5D_ALLOC_TIME_INCR`, or `H5D_ALLOC_TIME_LATE`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
dcpl = H5D.get_create_plist(dset_id);
alloc_time = H5P.get_alloc_time(dcpl);
switch(alloc_time)
    case H5ML.get_constant_value('H5D_ALLOC_TIME_DEFAULT')
        fprintf('allocation time is default\n');
    case H5ML.get_constant_value('H5D_ALLOC_TIME_EARLY')
        fprintf('allocation time is dataset creation time\n');
    case H5ML.get_constant_value('H5D_ALLOC_TIME_INCR')
        fprintf('allocation time is incremental\n');
    case H5ML.get_constant_value('H5D_ALLOC_TIME_LATE')
        fprintf('allocation time is when data is first written\n');
end
```

H5P.get_chunk

Purpose

Return size of chunks

Syntax

```
[rank,h5_chunk_dims] = H5P.get_chunk(plist_id)
```

Description

`[rank,h5_chunk_dims] = H5P.get_chunk(plist_id)` retrieves the size of chunks for the raw data of a chunked layout dataset for the dataset creation property list specified by `plist_id`.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_chunk_dims` parameter assumes C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g4/time');
dcpl = H5D.get_create_plist(dset_id);
[rank,chunk_dims] = H5P.get_chunk(dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.set_chunk`

H5P.get_external

Purpose

Return information about external file

Syntax

```
[name,offset,size] = H5P.get_external(plist_id,idx)
```

Description

`[name,offset,size] = H5P.get_external(plist_id,idx)` returns information about the external file specified by the dataset creation property list `plist_id`. The `idx` specifies the external file index, which is a number from zero to `N-1`, where `N` is the value returned by `H5P.get_external_count`. The `name` output returns the name of the external file (limited by 2048 characters). The `offset` output returns the location in bytes, from the beginning of the external file, where the data starts. The `size` output returns the size of the external data.

See Also

`H5P.get_external_count`

H5P.get_external_count

Purpose

Return count of external files

Syntax

```
num_files = H5P.get_external_count(plist_id)
```

Description

`num_files = H5P.get_external_count(plist_id)` returns the number of external files for the dataset creation property list, `plist_id`.

See Also

`H5P.get_external`

Purpose Return time when fill values are written to dataset

Syntax

```
fill_time = H5P.get_fill_time(plist_id)
```

Description

`fill_time = H5P.get_fill_time(plist_id)` returns the time when fill values are written to the dataset specified by the dataset creation property list `plist_id`. The `fill_time` output is one of the following values: `H5D_FILL_TIME_IFSET`, `H5D_FILL_TIME_ALLOC`, or `H5D_FILL_TIME_NEVER`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
dcpl = H5D.get_create_plist(dset_id);
fill_time = H5P.get_fill_time(dcpl);
switch(fill_time)
    case H5ML.get_constant_value('H5D_FILL_TIME_IFSET')
        fprintf('upon allocation if and only if fill value set by user
    case H5ML.get_constant_value('H5D_FILL_TIME_ALLOC')
        fprintf('written when storage space is allocated\n');
    case H5ML.get_constant_value('H5D_FILL_TIME_NEVER')
        fprintf('fill values are never written\n');
end
```

See Also

H5P.get_fill_time | H5P.set_fill_value

H5P.get_fill_value

Purpose

Return dataset fill value

Syntax

```
value = H5P.get_fill_value(plist_id,type_id)
```

Description

`value = H5P.get_fill_value(plist_id,type_id)` returns the dataset fill value defined in the dataset creation property list `plist_id`. The `type_id` input specifies the datatype of the returned fill value.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
dcpl = H5D.get_create_plist(dset_id);
type_id = H5T.copy('H5T_NATIVE_INT');
fill_value = H5P.get_fill_value(dcpl,type_id);
```

See Also

`H5P.set_fill_value` | `H5P.get_fill_time` | `H5P.set_fill_time`

H5P.get_filter

Purpose

Return information about filter in pipeline

Syntax

```
[filter,flags,cd_values,name] = H5P.get_filter(plist_id,
   index)
[filter,flags,cd_values,name,
   filter_config] = H5P.get_filter(plist_id,index)
```

Description

`[filter,flags,cd_values,name] = H5P.get_filter(plist_id,index)` returns information about the filter, specified by its filter index, in the filter pipeline, specified by the property list with which it is associated. This interface corresponds to the 1.6 version of `H5Pget_filter` in the HDF5 library.

`[filter,flags,cd_values,name,filter_config] = H5P.get_filter(plist_id,index)` returns information about the filter, specified by its filter index, in the filter pipeline, specified by the property list with which it is associated. It also returns information about the filter. Consult the HDF5 documentation for `H5Zget_filter_info` for information about `filter_config`. This interface corresponds to the 1.8 version of `H5Pget_filter` in the HDF5 library.

See Also

`H5P.get_nfilters` | `H5P.get_filter_by_id` | `H5P.modify_filter` | `H5P.remove_filter`

H5P.get_filter_by_id

Purpose

Return information about specified filter

Syntax

```
[flags,cd_values,name,
   filter_config] = H5P.get_filter_by_id(plist_id,idx)
```

Description

`[flags,cd_values,name,filter_config] = H5P.get_filter_by_id(plist_id,idx)` returns information about the filter specified by the filter id, `idx`.

See Also

`H5P.get_filter` | `H5P.get_nfilters` | `H5P.modify_filter` | `H5P.remove_filter`

Purpose

Determine layout of raw data for dataset

Syntax

```
layout = H5P.get_layout(dcpl)
```

Description

`layout = H5P.get_layout(dcpl)` returns the layout of the raw data for the dataset specified by the dataset creation property list, `dcpl`. Possible values are: `H5D_COMPACT`, `H5D_CONTIGUOUS`, or `H5D_CHUNKED`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
dcpl = H5D.get_create_plist(dset_id);
layout = H5P.get_layout(dcpl);
switch(layout)
    case H5ML.get_constant_value('H5D_COMPACT')
        fprintf('layout is compact\n');
    case H5ML.get_constant_value('H5D_CONTIGUOUS')
        fprintf('layout is contiguous\n');
    case H5ML.get_constant_value('H5D_CHUNKED')
        fprintf('layout is chunked\n');
end
```

See Also

`H5P.set_layout`

H5P.get_nfilters

Purpose

Return number of filters in pipeline

Syntax

```
num_filters = H5P.get_nfilters(plist_id)
```

Description

`num_filters = H5P.get_nfilters(plist_id)` returns the number of filters defined in the filter pipeline associated with the dataset creation property list, `plist_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g4/world');
dcpl = H5D.get_create_plist(dset_id);
num_filters = H5P.get_nfilters(dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.get_filter` | `H5P.get_filter_by_id` | `H5P.modify_filter` | `H5P.remove_filter`

Purpose

Modify filter in pipeline

Syntax

```
H5P.modify_filter(plist_id,filter_id,flags,cd_values)
```

Description

`H5P.modify_filter(plist_id,filter_id,flags,cd_values)` modifies the specified filter in the filter pipeline. `plist_id` is a property list identifier. `flags` is a bit vector specifying certain general properties of the filter. `cd_values` specifies auxiliary data for the filter.

See Also

`H5P.get_filter` | `H5P.get_nfilters` | `H5P.get_filter_by_id` | `H5P.remove_filter`

H5P.remove_filter

Purpose

Remove filter from property list

Syntax

```
H5P.remove_filter(plist_id,filter)
```

Description

`H5P.remove_filter(plist_id,filter)` removes the specified filter from the filter pipeline. `plist_id` is the dataset creation property list identifier.

See Also

`H5P.get_filter` | `H5P.get_nfilters` | `H5P.get_filter_by_id` | `H5P.modify_filter`

Purpose

Set timing for storage space allocation

Syntax

```
H5P.set_alloc_time(plist_id,alloc_time)
```

Description

`H5P.set_alloc_time(plist_id,alloc_time)` sets the timing for the allocation of storage space for a dataset's raw data. `plist_id` is a dataset creation property list. `alloc_time` can have any of the following values: `H5D_ALLOC_TIME_DEFAULT`, `H5D_ALLOC_TIME_EARLY`, `H5D_ALLOC_TIME_INC`, or `H5D_ALLOC_TIME_LATE`.

Examples

Create a 1000x500 double precision dataset with late allocation time.

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [1000 500];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = H5P.create('H5P_DATASET_CREATE');
alloc_time = H5ML.get_constant_value('H5D_ALLOC_TIME_LATE');
H5P.set_alloc_time(dcpl,alloc_time);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.get_alloc_time`

H5P.set_chunk

Purpose Set chunk size

Syntax

```
H5P.set_chunk(plist_id,h5_chunk_dims)
```

Description

`H5P.set_chunk(plist_id,h5_chunk_dims)` sets the size of the chunks used to store a chunked layout dataset. `plist_id` is a dataset creation property list identifier. `h5_chunk_dims` is an array specifying the size, in dataset elements, of each chunk.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_chunk_dims` parameter assumes C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Create a two dimensional double precision dataset that has an initial size of [512 1024], but is also unlimited in both dimensions and has a chunk size of [512 1024].

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
unlimited = H5ML.get_constant_value('H5S_UNLIMITED');
dims = [512 1024];
h5_dims = fliplr(dims);
h5_maxdims = [unlimited unlimited];
space_id = H5S.create_simple(2,[1024 512],h5_maxdims);
dcpl = H5P.create('H5P_DATASET_CREATE');
chunk_dims = [512 1024];
h5_chunk_dims = fliplr(chunk_dims);
H5P.set_chunk(dcpl,h5_chunk_dims);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also H5P.get_chunk

H5P.set_deflate

Purpose

Set compression method and compression level

Syntax

`H5P.set_deflate(plist_id,level)`

Description

`H5P.set_deflate(plist_id,level)` sets the compression method for the dataset creation property list specified by `plist_id` to `H5D_COMPRESS_DEFLATE`. `level` specifies the compression level as a value from 0 and 9, inclusive. Lower values results in less compression.

Examples

Create a two dimensional double precision dataset that has an initial size of [512 1024], but is also unlimited in both dimensions and has a chunk size of [512 1024] and a compression level of 5.

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
unlimited = H5ML.get_constant_value('H5S_UNLIMITED');
dims = [512 1024];
h5_dims = fliplr(dims);
h5_maxdims = [unlimited unlimited];
space_id = H5S.create_simple(2,[1024 512],h5_maxdims);
dcpl = H5P.create('H5P_DATASET_CREATE');
chunk_dims = [512 1024];
h5_chunk_dims = fliplr(chunk_dims);
H5P.set_chunk(dcpl,h5_chunk_dims);
H5P.set_deflate(dcpl,5);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

Purpose

Add additional file to external file list

Syntax

```
H5P.set_external(plist_id,name,offset,nbytes)
```

Description

`H5P.set_external(plist_id,name,offset,nbytes)` adds the external file specified by `name` to the list of external files in the dataset creation property list, `plist_id`. The `offset` argument specifies the location, in bytes, where the data starts relative to the beginning of the file. `nbytes` is the number of bytes reserved in the file for the data. `nbytes` may also be given as `'H5F_UNLIMITED'`, in which case the external file may be of unlimited size.

Examples

Create a dataset with an unlimited size external file.

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 50];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = H5P.create('H5P_DATASET_CREATE');
H5P.set_external(dcpl,'myexternalfile.dat',0,'H5F_UNLIMITED');
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
data = rand(dims);
dxpl = 'H5P_DEFAULT';
H5D.write(dset_id,'H5ML_DEFAULT','H5S_ALL','H5S_ALL',dxpl,data);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.get_external` | `H5ML.get_constant_value`

H5P.set_fill_time

Purpose

Set time when fill values are written to dataset

Syntax

```
H5P.set_fill_time(plist_id,fill_time)
```

Description

`H5P.set_fill_time(plist_id,fill_time)` sets the timing for writing fill values to a dataset in the dataset creation property list `plist_id`. The timing can be specified by one of the following values: `H5D_FILL_TIME_IFSET`, `H5D_FILL_TIME_ALLOC`, or `H5D_FILL_TIME_NEVER`.

Examples

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 50];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = H5P.create('H5P_DATASET_CREATE');
fill_time = H5ML.get_constant_value('H5D_FILL_TIME_ALLOC');
H5P.set_fill_time(dcpl,fill_time);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.get_fill_time` | `H5P.get_fill_value` | `H5P.set_fill_value`

Purpose

Set fill value for dataset creation property list

Syntax

```
H5P.set_fill_value(plist_id,type_id,value)
```

Description

`H5P.set_fill_value(plist_id,type_id,value)` sets the fill value for a the dataset creation property list specified by `plist_id`. The `value` argument specifies the fill value. `type_id` specifies the datatype of the fill value. Setting `value` to an empty array indicates that the fill value is to be undefined.

Examples

Create a double precision dataset with a fill value of -999.

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 50];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = H5P.create('H5P_DATASET_CREATE');
fill_time = H5ML.get_constant_value('H5D_FILL_TIME_ALLOC');
H5P.set_fill_time(dcpl,fill_time);
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
H5P.set_fill_value(dcpl,type_id,-999);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

H5P.set_filter

Purpose

Add filter to filter pipeline

Syntax

```
H5P.set_filter(plist_id,filter,flags,cd_values)
```

Description

`H5P.set_filter(plist_id,filter,flags,cd_values)` adds the specified filter and corresponding properties to the end of an output filter pipeline. `plist_id` is a property list identifier. `filter` is a filter identifier and should correspond to one of the following values:

```
H5P_FILTER_DEFLATE
H5P_FILTER_SHUFFLE
H5P_FILTER_FLETCHER32
```

`flags` is a bit vector specifying properties of the filter. `cd_values` is an array that contains auxiliary data for the filter.

See Also

`H5P.set_deflate` | `H5P.set_fletcher32` | `H5P.set_shuffle`

H5P.set_fletcher32

Purpose

Set Fletcher32 checksum filter in dataset creation

Syntax

`H5P.set_fletcher32(plist_id)`

Description

`H5P.set_fletcher32(plist_id)` sets the Fletcher32 checksum filter in the dataset creation property list specified by `plist_id`. The dataset creation property list must also have chunking enabled.

Examples

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,dims,[]);
dcpl = H5P.create('H5P_DATASET_CREATE');
chunk_dims = [10 20];
h5_chunk_dims = fliplr(chunk_dims);
H5P.set_chunk(dcpl,h5_chunk_dims);
H5P.set_fletcher32(dcpl);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.set_deflate` | `H5P.set_shuffle`

H5P.set_layout

Purpose

Set type of storage for dataset

Syntax

`H5P.set_layout(dcpl,layout)`

Description

`H5P.set_layout(dcpl,layout)` sets the type of storage used to store the raw data for the dataset creation property list, `dcpl`. The `layout` argument specifies the type of storage layout for raw data: `H5D_COMPACT`, `H5D_CONTIGUOUS`, or `H5D_CHUNKED`.

Examples

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,dims,[]);
dcpl = H5P.create('H5P_DATASET_CREATE');
layout = H5ML.get_constant_value('H5D_CONTIGUOUS');
H5P.set_layout(dcpl,layout);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.get_layout` | `H5P.set_chunk`

H5P.set_nbit

Purpose Set N-Bit filter

Syntax

```
H5P.set_nbit(plist_id)
```

Description `H5P.set_nbit(plist_id)` sets the N-Bit filter, `H5Z_FILTER_NBIT`, in the dataset creation property list `plist_id`.

H5P.set_scaleoffset

Purpose Set Scale-Offset filter

Syntax

```
H5P.set_scaleoffset(plistId,scaleType,scaleFactor)
```

Description `H5P.set_scaleoffset(plistId,scaleType,scaleFactor)` sets the Scale-Offset filter, `H5Z_FILTER_SCALEOFFSET`, for a dataset. For integer data types, the parameter `scaleType` should be set to the enumerated value `H5Z_SO_INT`. For floating-point data types, the `scaleType` should be the enumerated value `H5Z_SO_FLOAT_DSCALE`. Chunking must already be enabled on the dataset creation property list.

See Also `H5P.set_chunk`

Purpose

Set shuffle filter

Syntax

```
H5P.set_shuffle(plist_id)
```

Description

`H5P.set_shuffle(plist_id)` sets the shuffle filter, `H5Z_FILTER_SHUFFLE`, in the dataset creation property list `plist_id`. Compression must be enabled on the dataset creation property list in order to use the shuffle filter, and best results are usually obtained when the shuffle filter is set immediately prior to setting the deflate filter.

Examples

```
fid = H5F.create('myfile.h5');
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,dims,[]);
dcpl = H5P.create('H5P_DATASET_CREATE');
chunk_dims = [10 20];
h5_chunk_dims = fliplr(chunk_dims);
H5P.set_chunk(dcpl,h5_chunk_dims);
H5P.set_shuffle(dcpl);
H5P.set_deflate(dcpl,5);
dset_id = H5D.create(fid,'DS',type_id,space_id,dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.set_deflate`

H5P.get_alignment

Purpose Retrieve alignment properties

Syntax

```
[threshold,alignment] = H5P.get_alignment(plist_id)
```

Description

`[threshold,alignment] = H5P.get_alignment(plist_id)` retrieves the current settings for alignment properties from the file access property list specified by `plist_id`.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
[threshold,alignment] = H5P.get_alignment(fapl);
H5P.close(fapl);
H5F.close(fid);
```

H5P.get_driver

Purpose Low-level file driver

Syntax

```
driver_id = H5P.get_driver(plist_id)
```

Description

`driver_id = H5P.get_driver(plist_id)` returns the identifier of the low-level file driver associated with the file access property list or data transfer property list specified by `plist_id`. See HDF5 documentation for a list of valid return values.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
driver_id = H5P.get_driver(fapl);
if ( driver_id == H5ML.get_constant_value('H5FD_SEC2') )
    fprintf('File driver is H5FD_SEC2.\n');
end
H5P.close(fapl);
H5F.close(fid);
```

See Also `H5ML.get_constant_value`

H5P.get_family_offset

Purpose

Offset for family file driver

Syntax

```
offset = H5P.get_family_offset(fapl_id)
```

Description

`offset = H5P.get_family_offset(fapl_id)` retrieves the value of offset from the file access property list, `fapl_id`. `offset` is the offset of the data in the HDF5 file that is stored on disk in the selected member file in a family of files.

See Also

`H5P.set_family_offset`

Purpose

Information about core file driver properties

Syntax

```
[increment,backing_store] = H5P.get_fapl_core(fapl_id)
```

Description

`[increment,backing_store] = H5P.get_fapl_core(fapl_id)` queries the `H5FD_CORE` driver properties as set by `H5P.set_fapl_core`. The `fapl_id` argument specifies a file access property list. The return value `increment` specifies the size, in bytes, of memory increments. `backing_store` is a Boolean flag indicating whether to write the file contents to disk when the file is closed.

See Also

`H5P.set_fapl_core`

H5P.get_fapl_family

Purpose

File access property list information

Syntax

```
[memb_size,memb_fapl_id] = H5P.get_fapl_family(fapl_id)
```

Description

`[memb_size,memb_fapl_id] = H5P.get_fapl_family(fapl_id)` returns the size in bytes of each file member and the identifier of the file access property list for use with the family driver specified by `fapl_id`.

See Also

`H5P.set_fapl_family`

H5P.get_fapl_multi

Purpose Information about multifile access property list

Syntax

```
[memb_map,memb_fapl,memb_name,memb_addr,
   relax] = H5P.get_fapl_multi(fapl_id)
```

Description `[memb_map,memb_fapl,memb_name,memb_addr,relax] = H5P.get_fapl_multi(fapl_id)` returns information about the multifile access property list specified by `fapl_id`. The `memb_map` output maps memory usage types to other memory usage types. `memb_fapl` is a property list for each memory usage type. `memb_name` is the name generator for names of member files. `relax` is a Boolean value that, when non-zero, allows read-only access to incomplete file sets.

See Also `H5P.set_fapl_multi`

H5P.get_fclose_degree

Purpose

File close degree

Syntax

```
degree = H5P.get_fclose_degree(fapl_id)
```

Description

`degree = H5P.get_fclose_degree(fapl_id)` returns the current setting of the file close degree property `fc_degree` in the file access property list specified by `fapl_id`. Possible return values are: `H5F_CLOSE_DEFAULT`, `H5F_CLOSE_WEAK`, `H5F_CLOSE_SEMI`, or `H5F_CLOSE_STRONG`.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
degree = H5P.get_fclose_degree(fapl);
switch(degree)
    case H5ML.get_constant_value('H5F_CLOSE_DEFAULT')
        fprintf('file close degree is default\n');
    case H5ML.get_constant_value('H5F_CLOSE_WEAK')
        fprintf('file close degree is weak\n');
    case H5ML.get_constant_value('H5F_CLOSE_SEMI')
        fprintf('close degree is semi\n');
    case H5ML.get_constant_value('H5F_CLOSE_STRONG')
        fprintf('close degree is strong\n');
end
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5P.set_fclose_degree`

Purpose

Library version bounds settings

Syntax

```
[low,high] = H5P.get_libver_bounds(fapl_id)
```

Description

`[low,high] = H5P.get_libver_bounds(fapl_id)` gets bounds on library version bounds settings that control the format versions used when creating objects in the file with access property list `fapl_id`.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
[low,high] = H5P.get_libver_bounds(fapl);
```

See Also

`H5F.get_access_plist` | `H5P.set_libver_bounds`

H5P.get_gc_references

Purpose

Garbage collection references setting

Syntax

```
gc_ref = H5P.get_gc_references(fapl_id)
```

Description

`gc_ref = H5P.get_gc_references(fapl_id)` returns the current setting for the garbage collection references property from the file access property list specified by `fapl_id`. If `gc_ref` is `1`, garbage collection is on; if `0`, garbage collection is off.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
gc_ref = H5P.get_gc_references(fapl);
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5P.set_gc_references`

Purpose Metadata cache configuration

Syntax

```
config_struct = H5P.get_mdc_config(plist_id)
```

Description

`config_struct = H5P.get_mdc_config(plist_id)` returns the current metadata cache configuration from the indicated file access property list.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
config = H5P.get_mdc_config(fapl);
H5P.close(fapl);
H5F.close(fid);
```

See Also `H5P.set_mdc_config`

H5P.get_meta_block_size

Purpose

Metadata block size setting

Syntax

```
sz = H5P.get_meta_block_size(fapl_id)
```

Description

`sz = H5P.get_meta_block_size(fapl_id)` returns the current minimum size, in bytes, of new metadata block allocations.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
sz = H5P.get_meta_block_size(fapl);
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5P.set_meta_block_size`

Purpose

Type of data property for MULTI driver

Syntax

```
type = H5P.get_multi_type(fapl_id)
```

Description

`type = H5P.get_multi_type(fapl_id)` returns the type of data setting from the file access or data transfer property list, `fapl_id`.

This function should only be used with an HDF5 file written as a set of files with the MULTI file driver.

See Also

`H5P.set_multi_type`

H5P.get_sieve_buf_size

Purpose

Maximum data sieve buffer size

Syntax

```
sz = H5P.get_sieve_buf_size(fapl_id)
```

Description

`sz = H5P.get_sieve_buf_size(fapl_id)` returns the current maximum size of the data sieve buffer.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
sz = H5P.get_sieve_buf_size(fapl);
H5P.close(fapl);
H5F.close(fid);
```

See Also

`H5P.set_sieve_buf_size`

Purpose Small data block size setting

Syntax

```
sz = H5P.get_small_data_block_size(fapl_id)
```

Description `sz = H5P.get_small_data_block_size(fapl_id)` returns the current setting for the size of the small data block. `fapl_id` is a file access property list identifier.

Examples

```
fid = H5F.open('example.h5');
fapl = H5F.get_access_plist(fid);
sz = H5P.get_small_data_block_size(fapl);
H5P.close(fapl);
H5F.close(fid);
```

See Also `H5P.set_small_data_block_size`

H5P.set_alignment

Purpose

Set alignment properties for file access property list

Syntax

```
H5P.set_alignment(fapl_id,threshold,alignment)
```

Description

`H5P.set_alignment(fapl_id,threshold,alignment)` sets the alignment properties of the file access property list specified by `fapl_id` so that any file object greater than or equal in size to threshold (in bytes) is aligned on an address which is a multiple of alignment.

In most cases the default values of threshold and alignment result in the best performance.

See Also

`H5P.get_alignment`

Purpose Set offset property for family of files

Syntax

```
H5P.set_family_offset(fapl_id,offset)
```

Description `H5P.set_family_offset(fapl_id,offset)` sets offset property in the file access property list specified by `fapl_id` for low-level access to a file in a family of files. `offset` identifies a user-determined location from the beginning of the HDF5 file in bytes.

See Also `H5P.get_family_offset`

H5P.set_fapl_core

Purpose

Modify file access to use H5FD_CORE driver

Syntax

```
H5P.set_fapl_core(fapl_id,increment,backing_store)
```

Description

`H5P.set_fapl_core(fapl_id,increment,backing_store)` modifies the file access property list to use the H5FD_CORE driver. `increment` specifies the increment by which allocated memory is to be increased each time more memory is required. `backing_store` is a Boolean flag that, when non-zero, indicates the file contents should be written to disk when the file is closed.

Examples

Create a file image in memory only.

```
plist = 'H5P_DEFAULT';
ndatasets = 20;
block_size = 1024*1024;
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_fapl_core(fapl,2^16,false);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',plist,fapl);
space_id = H5S.create_simple(1, block_size, []);
type_id = H5T.copy('H5T_IEEE_F64LE');
data = zeros(block_size,1);
for j = 1:ndatasets
 dsname = sprintf( 'dset%02d', j);
 fprintf( 'Writing dataset %s...\n',dsname);
 dsid = H5D.create(fid,dsname,type_id,space_id,'H5P_DEFAULT');
 H5D.write(dsid,'H5ML_DEFAULT',space_id,space_id,plist,data);
 H5D.close(dsid);
end
H5P.close(fapl);
H5S.close(space_id);
H5T.close(type_id);
H5F.close(fid);
dir('myfile.h5');
```

See Also

`H5P.get_fapl_core`

Purpose

Set file access to use family driver

Syntax

```
H5P.set_fapl_family(fapl_id,memb_size,memb_fapl_id)
```

Description

`H5P.set_fapl_family(fapl_id,memb_size,memb_fapl_id)` sets the file access property list, specified by `fapl_id`, to use the family driver. `memb_size` is the size in bytes of each file member. `memb_fapl_id` is the identifier of the file access property list to be used for each family member.

Examples

```
plist = 'H5P_DEFAULT';
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_fapl_family(fapl, 8192, plist);
fid = H5F.create('family%d.h5','H5F_ACC_TRUNC','H5P_DEFAULT',fapl);
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [50 25];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,[]);
dset_id = H5D.create(fid,'DS',type_id,space_id,plist)
data = reshape(1:prod(dims),dims);
H5D.write(dset_id,'H5ML_DEFAULT','H5S_ALL','H5S_ALL',plist,data);
H5P.close(fapl);
H5T.close(type_id);
H5S.close(space_id);
H5D.close(dset_id);
dir('*.h5');
h5disp('family%d.h5');
```

See Also

`H5P.get_fapl_family`

H5P.set_fapl_log

Purpose

Set use of logging driver

Syntax

```
H5P.set_fapl_log(fapl_id,logfile,flags,buf_size)
```

Description

`H5P.set_fapl_log(fapl_id,logfile,flags,buf_size)` modifies the file access property list, `fapl_id`, to use the logging driver H5FD_LOG. `logfile` is the name of the file in which the logging entries are to be recorded. `flags` is a bit mask that specifies the types of activity to log. See the HDF5 documentation for a list of available flag settings. `buf_size` specifies the size of the logging buffer.

Purpose

Set use of multifile driver

Syntax

```
H5P.set_fapl_multi(fapl_id,relax)
H5P.set_fapl_multi(fapl_id,memb_map,memb_fapl,memb_name,
   memb_addr,relax)
```

Description

`H5P.set_fapl_multi(fapl_id,relax)` sets the file access property list, `fapl_id`, to access HDF5 files created with the multi-driver with default values provided by the HDF5 library. `relax` is a Boolean value that allows read-only access to incomplete file sets when set to `1`.

`H5P.set_fapl_multi(fapl_id,memb_map,memb_fapl,memb_name,memb_addr,rela` sets the file access property list to use the multifile driver. `memb_map` maps memory usage types to other memory usage types. `memb_fapl` contains a property list for each memory usage type. `memb_name` is a name generator for names of member files. `memb_addr` specifies the offsets within the virtual address space at which each type of data storage begins.

See Also

`H5P.get_fapl_multi`

H5P.set_fapl_sec2

Purpose

Set file access for sec2 driver

Syntax

```
H5P.set_fapl_sec2(fapl_id)
```

Description

`H5P.set_fapl_sec2(fapl_id)` modifies the file access property list, `fapl_id`, to use the H5FD_SEC2 driver.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_fapl_sec2(fapl);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

Purpose

Set file access for emulation of split file driver

Syntax

```
H5P.set_fapl_split(fapl_id,meta_ext,meta_plist_id,raw_ext,
   raw_plist_id)
```

Description

`H5P.set_fapl_split(fapl_id,meta_ext,meta_plist_id,raw_ext,raw_plist_id)` is a compatibility function that enables the multi-file driver to emulate the split driver from HDF5 Releases 1.0 and 1.2. `meta_ext` is a text string that specifies the metadata filename extension. `meta_plist_id` is a file access property list identifier for the metadata file. `raw_ext` is a text string that specifies the raw data filename extension. `raw_plist_id` is the file access property list identifier for the raw data file.

H5P.set_fapl_stdio

Purpose

Set file access for standard I/O driver

Syntax

```
H5P.set_fapl_stdio(fapl_id)
```

Description

`H5P.set_fapl_stdio(fapl_id)` modifies the file access property list, `fapl_id`, to use the standard I/O driver, H5FD_STDIO.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_fapl_stdio(fapl);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

Purpose

Set file access for file close degree

Syntax

```
H5P.set_fclose_degree(fapl_id,degree)
```

Description

`H5P.set_fclose_degree(fapl_id,degree)` sets the file close degree property in the file access property list, `fapl_id`, to the value specified by `degree`. The `degree` argument can have any of the following values:

- 'H5F_CLOSE_WEAK'
- 'H5F_CLOSE_SEMI'
- 'H5F_CLOSE_STRONG'
- 'H5F_CLOSE_DEFAULT'

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_fclose_degree(fapl,'H5F_CLOSE_STRONG');
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_fclose_degree`

H5P.set_gc_references

Purpose

Set garbage collection references flag

Syntax

```
H5P.set_gc_references(fapl_id,gc_ref)
```

Description

`H5P.set_gc_references(fapl_id,gc_ref)` sets the flag for garbage collecting references for the file specified by the file access property list identifier, `fapl_id`. The `gc_ref` argument is a flag setting reference garbage collection to on (1) or off (0).

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_gc_references(fapl,1);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_gc_references`

Purpose

Set library version bounds for objects

Syntax

`H5P.get_libver_bounds(fapl_id,low,high)`

Description

`H5P.get_libver_bounds(fapl_id,low,high)` sets bounds on library versions, and indirectly format versions, to be used when creating objects in the file with access property list `fapl_id`. The `low` argument must be set to either of `'H5F_LIBVER_EARLIEST'`, `'H5F_LIBVER_18'` or `'H5F_LIBVER_LATEST'`. The `high` argument must be set to `'H5F_LIBVER_18'` or `'H5F_LIBVER_LATEST'`.

Examples

Create an HDF5 file where objects are created using the latest available format for each object.

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_libver_bounds(fapl,'H5F_LIBVER_LATEST','H5F_LIBVER_LATEST');
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
```

See Also

`H5P.get_libver_bounds | H5ML.get_constant_value`

H5P.set_mdc_config

Purpose

Set initial metadata cache configuration

Syntax

```
H5P.set_mdc_config(plist_id,config_struct)
```

Description

`H5P.set_mdc_config(plist_id,config_struct)` sets the initial metadata cache configuration in the indicated file access property list to the supplied values. Before using this function, you should retrieve the current configuration using `H5P.get_mdc_config`.

Many of the fields in the structure, `config_struct`, are intended to be used only in close consultation with the HDF5 Group itself.

See Also

`H5P.get_mdc_config`

H5P.set_meta_block_size

Purpose

Set minimum metadata block size

Syntax

```
H5P.set_meta_block_size(fapl_id,size)
```

Description

`H5P.set_meta_block_size(fapl_id,size)` sets the minimum metadata block size for the file access property list specified by `fapl_id`. The `size` argument specifies minimum size, in bytes, of metadata block allocations.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_meta_block_size(fapl,4096);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_meta_block_size`

H5P.set_multi_type

Purpose

Sepcify type of data accessed with MULTI driver

Syntax

```
H5P.set_multi_type(fapl_id,type)
```

Description

`H5P.set_multi_type(fapl_id,type)` sets the type of data property in the file access or data transfer property list `fapl_id`. The `type` argument can have any of the following values: `H5FD_MEM_SUPER`, `H5FD_MEM_BTREE`, `H5FD_MEM_DRAW`, `H5FD_MEM_GHEAP`, `H5FD_MEM_LHEAP`, or `H5FD_MEM_OHDR`.

See Also

`H5P.get_multi_type`

H5P.set_sieve_buf_size

Purpose Set maximum size of data sieve buffer

Syntax

```
H5P.set_sieve_buf_size(fapl_id,buffer_size)
```

Description `H5P.set_sieve_buf_size(fapl_id,buffer_size)` sets `buffer_size`, the maximum size in bytes of the data sieve buffer, which is used by file drivers that are capable of using data sieving. `fapl_id` is a file access property list identifier.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_sieve_buf_size(fapl,131072);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also `H5P.get_sieve_buf_size`

H5P.set_small_data_block_size

Purpose

Set size of block reserved for small data

Syntax

```
H5P.set_small_data_block_size(fapl_id,size)
```

Description

`H5P.set_small_data_block_size(fapl_id,size)` sets the maximum size, in bytes, of a contiguous block reserved for small data. `fapl_id` is a file access property list identifier.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_small_data_block_size(fapl,4096);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_small_data_block_size`

Purpose

Return 1/2 rank of indexed storage B-tree

Syntax

```
ik = H5P.get_istore_k(plist_id)
```

Description

`ik = H5P.get_istore_k(plist_id)` returns the chunked storage B-tree 1/2 rank of the file creation property list specified by `plist_id`.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
ik = H5P.get_istore_k(fcpl);
H5P.close(fcpl);
H5F.close(fid);
```

See Also

`H5P.set_istore_k`

H5P.get_sizes

Purpose

Return size of offsets and lengths

Syntax

```
[sizeof_addr,sizeof_size] = H5P.get_sizes(fcpl)
```

Description

`[sizeof_addr,sizeof_size] = H5P.get_sizes(fcpl)` returns the size of the offsets and lengths used in an HDF5 file. `fcpl` specifies a file creation property list.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
[soaddr, sosize] = H5P.get_sizes(fcpl);
H5P.close(fcpl);
H5F.close(fid);
```

See Also

`H5P.set_sizes`

Purpose

Return size of B-tree 1/2 rank and leaf node 1/2 size

Syntax

```
[ik,lk] = H5P.get_sym_k(plist_id)
```

Description

`[ik,lk] = H5P.get_sym_k(plist_id)` returns the size of the symbol table B-tree 1/2 rank, `ik`, and the symbol table leaf node 1/2 size, `lk`. The `plist_id` argument is a file creation property list identifier.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
[ik, lk] = H5P.get_sym_k(fcpl);
H5P.close(fcpl);
H5F.close(fid);
```

See Also

`H5P.set_sym_k`

H5P.get_userblock

Purpose

Return size of user block

Syntax

```
sz = H5P.get_userblock(plist_id)
```

Description

`sz = H5P.get_userblock(plist_id)` returns the size of a user block in a file creation property list. `plist_id` is a property list identifier.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
sz = H5P.get_userblock(fcpl);
H5P.close(fcpl);
H5F.close(fid);
```

See Also

`H5P.set_userblock`

Purpose

Return version information for file creation property list

Syntax

```
[superblock,freelist,stab,shhdr] = H5P.get_version(fcpl)
```

Description

`[superblock,freelist,stab,shhdr] = H5P.get_version(fcpl)` returns the version of the super block, the global freelist, the symbol table, and the shared object header. Retrieving this information requires the file creation property list.

Examples

```
fid = H5F.open('example.h5');
fcpl = H5F.get_create_plist(fid);
[super,freelist,stab,shhdr] = H5P.get_version(fcpl);
H5P.close(fcpl);
H5F.close(fid);
```

H5P.set_istore_k

Purpose

Set size of parameter for indexing chunked datasets

Syntax

```
H5P.set_istore_k(plist_id,ik)
```

Description

`H5P.set_istore_k(plist_id,ik)` sets the size of the parameter used to control the B-trees for indexing chunked datasets for the file creation property list specified by `plist_id`. The `ik` argument is one half the rank of a tree that stores chunked raw data.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_istore_k(fcpl,64);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_istore_k`

Purpose

Set byte size of offsets and lengths

Syntax

```
H5P.set_sizes(plist_id,sizeof_addr,sizeof_size)
```

Description

`H5P.set_sizes(plist_id,sizeof_addr,sizeof_size)` sets the byte size of the offsets and lengths used to address objects in an HDF5 file. `plist_id` is a file creation property list.

See Also

`H5P.get_sizes`

H5P.set_sym_k

Purpose

Set size of parameters used to control symbol table nodes

Syntax

```
H5P.set_sym_k(plist_id,ik,lk)
```

Description

`H5P.set_sym_k(plist_id,ik,lk)` sets the size of parameters used to control the symbol table nodes for the file access property list, `plist_id`. The `ik` argument is one half the rank of a tree that stores a symbol table for a group. `lk` is one half of the number of symbols that can be stored in a symbol table node.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_sym_k(fcpl,32,8);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also

`H5P.get_sym_k`

H5P.set_userblock

Purpose Set user block size

Syntax

```
H5P.set_userblock(plist_id,size)
```

Description `H5P.set_userblock(plist_id,size)` sets the user block size of the file creation property list, `plist_id`.

Examples

```
fcpl = H5P.create('H5P_FILE_CREATE');
fapl = H5P.create('H5P_FILE_ACCESS');
H5P.set_userblock(fcpl,4096);
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',fcpl,fapl);
H5F.close(fid);
```

See Also `H5P.get_userblock`

H5P.get_attr_creation_order

Purpose Return tracking order and indexing settings

Syntax

```
crt_order_flags = H5P.get_attr_phase_change(ocpl_id)
```

Description

`crt_order_flags = H5P.get_attr_phase_change(ocpl_id)` retrieves tracking and indexing settings for attribute creation order. If `crt_order_flags` is zero, then the attribute creation order is neither tracked or indexed. Otherwise the creation order flags should be one of the following constant values:

- `H5P_CRT_ORDER_TRACKED`
- `H5P_CRT_ORDER_INDEXED`

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
dcpl = H5D.get_create_plist(dset_id);
flags = H5P.get_attr_creation_order(dcpl);
switch ( flags )
    case 0
        fprintf('neither tracked nor indexed\n');
    case H5ML.get_constant_value('H5P_CRT_ORDER_TRACKED')
        fprintf('tracked\n');
    case H5ML.get_constant_value('H5P_CRT_ORDER_INDEXED')
        fprintf('indexed\n');
end
H5P.close(dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.set_attr_creation_order` | `H5ML.get_constant_value`

H5P.get_attr_phase_change

Purpose

Retrieve attribute phase change thresholds

Syntax

```
[max_compact,min_dense] = H5P.get_attr_phase_change(ocpl_id)
```

Description

`[max_compact,min_dense] = H5P.get_attr_phase_change(ocpl_id)` retrieves attribute phase change thresholds for the dataset or group with creation property list `ocpl_id`.

`max_compact` is the maximum number of attributes to be stored in compact storage (default is 8).

`min_dense` is the minimum number of attributes to be stored in dense storage (default is 6).

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
dcpl = H5D.get_create_plist(dset_id);
[max_compact,min_dense] = H5P.get_attr_phase_change(dcpl);
H5P.close(dcpl);
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5P.set_attr_phase_change`

H5P.get_copy_object

Purpose

Return properties to be used when object is copied

Syntax

```
copy_options = H5P.get_copy_object(ocpl_id)
```

Description

`copy_options = H5P.get_copy_object(ocpl_id)` retrieves the properties currently specified in the object copy property list `ocpl_id`, which will be invoked when a new copy is made of an existing object.

Examples

```
ocpl = H5P.create('H5P_OBJECT_COPY');
options = H5P.get_copy_object(ocpl);
```

See Also

`H5P.set_copy_object`

H5P.set_attr_creation_order

Purpose

Set tracking of attribute creation order

Syntax

```
H5P.set_attr_creation_order(gcplId,crt_order_flags)
```

Description

`H5P.set_attr_creation_order(gcplId,crt_order_flags)` sets tracking and indexing of attribute creation order. The creation order flags should be either `H5P_CRT_ORDER_TRACKED` or a bitwise-or of `H5P_CRT_ORDER_TRACKED` and `H5P_CRT_ORDER_INDEXED`.

The default behavior is that attribute creation order is neither tracked nor indexed.

Examples

```
dcpl = H5P.create('H5P_DATASET_CREATE');
order = H5ML.get_constant_value('H5P_CRT_ORDER_TRACKED');
H5P.set_attr_creation_order(dcpl,order);
```

See Also

`H5P.get_attr_creation_order` | `H5ML.get_constant_value` | `bitor`

H5P.set_attr_phase_change

Purpose

Set attribute storage phase change thresholds

Syntax

```
H5P.get_attr_phase_change(ocpl_id,max_compact,min_dense)
```

Description

`H5P.get_attr_phase_change(ocpl_id,max_compact,min_dense)` sets attribute storage phase change thresholds for the group or dataset with creation order property list `ocpl_id`.

`max_compact` is the maximum number of attributes to be stored in compact storage (default is 8).

`min_dense` is the minimum number of attributes to be stored in dense storage (default is 6).

See Also

`H5P.get_attr_phase_change`

Purpose

Set properties to be used when objects are copied

Syntax

```
H5P.set_copy_object(ocp_plist_id,copy_options)
```

Description

`H5P.set_copy_object(ocp_plist_id,copy_options)` sets the properties in the object copy property list `ocp_plist_id` that will be invoked when a new copy is made of an existing object. `ocp_plist_id` is the object copy property list and specifies the properties governing the copying of the object.

Several flags, described below, are available for inclusion in the object copy property list:

`H5O_COPY_SHALLOW_HIERARCHY_FLAG`	Copy only immediate members of a group. Default behavior, without flag: Recursively copy all objects below the group.
`H5O_COPY_EXPAND_SOFT_LINK_FLAG`	Expand soft links into new objects. Default behavior, without flag: Keep soft links as they are.
`H5O_COPY_EXPAND_EXT_LINK_FLAG`	Expand external link into new objects. Default behavior, without flag: Keep external links as they are.
`H5O_COPY_EXPAND_REFERENCE_FLAG`	Copy objects that are pointed to by references. Default behavior, without flag: Update only the values of object references.
`H5O_COPY_WITHOUT_ATTR_FLAG`	Copy object without copying attributes. Default behavior, without flag: Copy object along with all its attributes.

Examples

```
ocp_plist_id = H5P.create ('H5P_OBJECT_COPY');
option1 = H5ML.get_constant_value('H5O_COPY_EXPAND_SOFT_LINK_FLAG');
option2 = H5ML.get_constant_value('H5O_COPY_EXPAND_REFERENCE_FLAG');
copy_options = bitor(option1,option2);
H5P.set_copy_object(ocp_plist_id, copy_options);
```

H5P.get_create_intermediate_group

Purpose

Determine creation of intermediate groups

Syntax

```
bool = H5P.get_create_intermediate_group(lcpl_id)
```

Description

`bool = H5P.get_create_intermediate_group(lcpl_id)` determines whether the link creation property list `lcpl_id` is set to enable creating missing intermediate groups.

Examples

```
lcpl = H5P.create('H5P_LINK_CREATE');
if H5P.get_create_intermediate_group(lcpl)
    fprintf('set to enable creating intermediate groups\n');
else
    fprintf('not set to enable creating intermediate groups\n');
end
```

See Also

`H5P.set_create_intermediate_group`

H5P.get_link_creation_order

Purpose

Query if link creation order is tracked

Syntax

```
crt_order_flags = H5P.get_link_phase_change(gcpl_id)
```

Description

`crt_order_flags = H5P.get_link_phase_change(gcpl_id)` queries whether link creation order is tracked or indexed in a group with creation property list identifier `gcpl_id`. The creation order flags should be one of the following constant values:

- `H5P_CRT_ORDER_TRACKED`
- `H5P_CRT_ORDER_INDEXED`

Examples

```
tracked = H5ML.get_constant_value('H5P_CRT_ORDER_TRACKED');
indexed = H5ML.get_constant_value('H5P_CRT_ORDER_INDEXED');
gcpl = H5P.create('H5P_GROUP_CREATE');
order = H5P.get_link_creation_order(gcpl);
if bitand(order,tracked)
    fprintf('order is tracked\n');
end
if bitand(order,indexed)
    fprintf('order is indexed\n');
end
```

See Also

`H5P.set_link_creation_order` | `H5ML.get_constant_value` | `bitand`

Purpose

Query settings for conversion between groups

Syntax

```
[max_compact,min_dense] = H5P.get_link_phase_change(gcpl_id)
```

Description

`[max_compact,min_dense] = H5P.get_link_phase_change(gcpl_id)` retrieves the settings for conversion between compact and dense groups.

`max_compact` is the maximum number of links to store as header messages in the group header before converting the group to the dense format. Groups that are in the compact format and exceed this number of links are automatically converted to the dense format.

`min_dense` is the minimum number of links to store in the dense format. Groups which are in dense format and in which the number of links falls below this number are automatically converted back to the compact format.

Examples

```
gcpl = H5P.create('H5P_GROUP_CREATE');
[max_compact, min_dense] = H5P.get_link_phase_change(gcpl);
```

See Also

`H5P.set_link_phase_change`

H5P.set_create_intermediate_group

Purpose

Set creation of intermediate groups

Syntax

```
H5P.set_create_intermediate_group(lcpl_id,flag)
```

Description

`H5P.set_create_intermediate_group(lcpl_id,flag)` specifies in the link creation property list `lcpl_id` whether to create missing intermediate groups.

Examples

Enable the creation of intermediate groups.

```
fid = H5F.create('myfile.h5');
lcpl = H5P.create('H5P_LINK_CREATE');
H5P.set_create_intermediate_group(lcpl,1);
gid = H5G.create(fid,'/a/b/c/d',lcpl,'H5P_DEFAULT','H5P_DEFAULT');
H5G.close(gid);
H5F.close(fid);
```

See Also

`H5P.get_create_intermediate_group`

Purpose

Set creation order tracking and indexing

Syntax

```
H5P.set_link_creation_order(gcplId,crt_order_flags)
```

Description

`H5P.set_link_creation_order(gcplId,crt_order_flags)` sets creation order tracking and indexing for links in the group with group creation property list `gcpl_id`.

The creation order flags should be one of the following constant values:

- `H5P_CRT_ORDER_TRACKED`
- `H5P_CRT_ORDER_INDEXED`

If only `H5P_CRT_ORDER_TRACKED` is set, HDF5 will track link creation order in any group created with the group creation property list `gcpl_id`. If both `H5P_CRT_ORDER_TRACKED` and `H5P_CRT_ORDER_INDEXED` are set, HDF5 will track link creation order in the group and index links on that property.

Examples

```
tracked = H5ML.get_constant_value('H5P_CRT_ORDER_TRACKED');
indexed = H5ML.get_constant_value('H5P_CRT_ORDER_INDEXED');
order = bitor(tracked,indexed);
gcpl = H5P.create('H5P_GROUP_CREATE');
H5P.set_link_creation_order(gcpl,order);
```

See Also

`H5P.get_link_creation_order` | `H5ML.get_constant_value`

H5P.set_link_phase_change

Purpose

Set parameters for group conversion

Syntax

```
H5P.get_link_phase_change(gcpl_id,max_compact,min_dense)
```

Description

`H5P.get_link_phase_change(gcpl_id,max_compact,min_dense)` sets the parameters for conversion between compact and dense groups.

`max_compact` is the maximum number of links to store as header messages in the group header before converting the group to the dense format. Groups that are in the compact format and exceed this number of links are automatically converted to the dense format.

`min_dense` is the minimum number of links to store in the dense format. Groups which are in dense format and in which the number of links falls below this number are automatically converted back to the compact format.

Examples

```
gcpl = H5P.create('H5P_GROUP_CREATE');
H5P.set_link_phase_change(gcpl,10,8);
```

See Also

`H5P.get_link_phase_change`

H5P.get_char_encoding

Purpose Return character encoding

Syntax

```
encoding = H5P.get_char_encoding(propertyList)
```

Description

`encoding = H5P.get_char_encoding(propertyList)` retrieves the character encoding used to encode strings or object names that are created with the property list `propertyList`. The values returned correspond to either `H5T_CSET_ASCII` or `H5T_CSET_UTF8`.

See Also

`H5P.set_char_encoding` | `H5ML.get_constant_value`

H5P.set_char_encoding

Purpose

Set character encoding used to encode strings

Syntax

```
H5P.set_char_encoding(propList,encoding)
```

Description

`H5P.set_char_encoding(propList,encoding)` sets the character encoding used to encode strings or object names that are created with the property list `propList`. The values of `encoding` should either be `H5T_CSET_ASCII` or `H5T_CSET_UTF8`.

See Also

`H5P.get_char_encoding` | `H5ML.get_constant_value`

Purpose

Create reference

Syntax

```
ref = H5R.create(loc_id,name,ref_type,space_id)
```

Description

`ref = H5R.create(loc_id,name,ref_type,space_id)` creates the reference, `ref`, of the type specified in `ref_type`, pointing to the object specified by name located at `loc_id`. The `ref_type` argument can be either `'H5R_OBJECT'`, or `'H5R_DATASET_REGION'`. The `space_id` argument should be `-1`, if `ref_type` is `'H5R_OBJECT'`.

Examples

Create a double-precision dataset and a reference dataset.

```
fid = H5F.create('myfile.h5');
type1_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [10 5];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space1_id = H5S.create_simple(2,h5_dims,h5_maxdims);
dcpl = 'H5P_DEFAULT';
dset1_id = H5D.create(fid,'my_double',type1_id,space1_id,dcpl);
type2_id = 'H5T_STD_REF_OBJ';
space2_id = H5S.create('H5S_SCALAR');
dset2_id = H5D.create(fid,'my_ref',type2_id,space2_id,dcpl);
ref_data = H5R.create(fid,'my_double','H5R_OBJECT',-1);
dxpl = 'H5P_DEFAULT';
H5D.write(dset2_id,'H5ML_DEFAULT','H5S_ALL','H5S_ALL',dxpl,ref_data);
H5D.close(dset1_id);
H5D.close(dset2_id);
H5F.close(fid);
```

See Also

`H5D.create`

H5R.dereference

Purpose Open object specified by reference

Syntax

```
output = H5R.dereference(dataset,ref_type,ref)
```

Description

`output = H5R.dereference(dataset,ref_type,ref)` returns an identifier to the object specified by `ref` in the dataset specified by `dataset`.

Examples

```
plist = 'H5P_DEFAULT';
space = 'H5S_ALL';
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/reference');
ref_data = H5D.read(dset_id,'H5T_STD_REF_OBJ',space,space,plist);
deref_dset_id = H5R.dereference(dset_id,'H5R_OBJECT',ref_data(:,1));
H5D.close(dset_id);
H5D.close(deref_dset_id);
H5F.close(fid);
```

See Also

`H5R.create` | `H5I.get_name`

Purpose Name of referenced object

Syntax

```
name = H5R.get_name(loc_id,ref_type,ref)
```

Description

`name = H5R.get_name(loc_id,ref_type,ref)` retrieves the name for the object identified by `ref`. The `loc_id` argument is the identifier for the dataset containing the reference or for the group containing that dataset. `ref_type` specifies the type of the reference `ref`. Valid values for `ref_type` are `'H5R_OBJECT'` or `'H5R_DATASET_REGION'`.

Examples

```
plist = 'H5P_DEFAULT';
space = 'H5S_ALL';
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/reference');
ref_data = H5D.read(dset_id,'H5T_STD_REF_OBJ',space,space,plist);
name = H5R.get_name(dset_id,'H5R_OBJECT',ref_data(:,1));
H5D.close(dset_id);
H5F.close(fid);
```

See Also `H5I.get_name`

H5R.get_obj_type

Purpose

Type of referenced object

Syntax

```
obj_type = H5R.get_obj_type(id,ref_type,ref)
```

Description

`obj_type = H5R.get_obj_type(id,ref_type,ref)` returns the type of object that an object reference points to. Valid values for `ref_type` are: `H5R_OBJECT` or `H5R_DATASET_REGION`. Valid return values correspond to the following values.

`'H5O_TYPE_GROUP'`	Object is a group.
`'H5O_TYPE_DATASET'`	Object is a dataset.
`'H5O_TYPE_NAMED_DATATYPE'`	Object is a named datatype.

This function corresponds to the 1.8 interface version of `H5Rget_obj_type` in the HDF5 library C API.

Examples

```
plist = 'H5P_DEFAULT';
space = 'H5S_ALL';
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/reference');
ref_data = H5D.read(dset_id,'H5T_STD_REF_OBJ',space,space,plist);
obj_type = H5R.get_obj_type(fid,'H5R_OBJECT',ref_data(:,1));
switch(obj_type)
    case H5ML.get_constant_value('H5O_TYPE_GROUP')
        fprintf('group\n');
    case H5ML.get_constant_value('H5O_TYPE_DATASET')
        fprintf('dataset\n');
    case H5ML.get_constant_value('H5O_TYPE_NAMED_DATATYPE')
        fprintf('named datatype\n');
end
H5D.close(dset_id);
H5F.close(fid);
```

See Also

`H5ML.get_constant_value`

Purpose

Copy of data space of specified region

Syntax

```
space_id = H5R.get_region(dataset,ref_type,ref)
```

Description

`space_id = H5R.get_region(dataset,ref_type,ref)` returns a data space with the specified region selected. `dataset` is used to identify the file containing the referenced region and can be any identifier for any object in the file.

Examples

```
space = 'H5S_ALL';
plist = 'H5P_DEFAULT';
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/region_reference');
ref_data = H5D.read(dset_id,'H5T_STD_REF_DSETREG',space,space,plist);
space_id = H5R.get_region(fid,'H5R_DATASET_REGION',ref_data(:,1));
H5S.close(space_id);
H5D.close(dset_id);
H5F.close(fid);
```

H5S.copy

Purpose

Create copy of data space

Syntax

```
output = H5S.copy(space_id)
```

Description

`output = H5S.copy(space_id)` creates a new data space, which is an exact copy of the dataspace identified by `space_id`. The `output` argument is a data space identifier.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g2/dset2.1');
space1_id = H5D.get_space(dset_id);
space2_id = H5S.copy(space1_id);
[~,dims1] = H5S.get_simple_extent_dims(space1_id)
[~,dims2] = H5S.get_simple_extent_dims(space2_id)
```

See Also

`H5D.get_space` | `H5S.get_simple_extent_dims`

Purpose

Create new data space

Syntax

```
space_id = H5S.create(space_type)
```

Description

`space_id = H5S.create(space_type)` creates a new dataspace of the type specified by `space_type`, which can be specified by one of the following strings.

`'H5S_SCALAR'`

`'H5S_SIMPLE'`

`'H5S_NULL'`

`space_id` is the identifier for the new dataspace.

Examples

Create a scalar dataspace.

```
space_id = H5S.create('H5S_SCALAR');
numpoints = H5S.get_simple_extent_npoints(space_id);
```

See Also

`H5S.get_simple_extent_npoints`

H5S.close

Purpose

Close data space

Syntax

```
H5S.close(space_id)
```

Description

`H5S.close(space_id)` releases and terminates access to a data space. `space_id` is a data space identifier.

See Also

`H5A.get_space` | `H5D.get_space`

H5S.create_simple

Purpose

Create new simple data space

Syntax

```
space_id = H5S.create_simple(rank,h5_dims,h5_maxdims)
```

Description

`space_id = H5S.create_simple(rank,h5_dims,h5_maxdims)` creates a new simple data space and opens it for access. `rank` is the number of dimensions used in the data space. `h5_dims` is an array specifying the size of each dimension of the dataset. `h5_maxdims` is an array specifying the upper limit on the size of each dimension. `space_id` is a data space identifier.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_dims` and `h5_maxdims` parameters assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Create a data space for a dataset with 10 rows and 5 columns.

```
dims = [10 5];
h5_dims = fliplr(dims);
h5_maxdims = h5_dims;
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
```

Create a data space for a dataset with 10 rows and 5 columns such that the dataset is extendible along the column dimension.

```
dims = [10 5];
h5_dims = fliplr(dims);
maxdims = [10 H5ML.get_constant_value('H5S_UNLIMITED')];
h5_maxdims = fliplr(maxdims);
space_id = H5S.create_simple(2,h5_dims,h5_maxdims);
```

See Also

H5S.create | H5S.close | H5ML.get_constant_value

Purpose

Copy extent from source to destination data space

Syntax

```
H5S.extent_copy(dst_id,src_id)
```

Description

`H5S.extent_copy(dst_id,src_id)` copies the extent from the source data space, `src_id`, to the destination data space, `dst_id`.

Examples

```
space_id1 = H5S.create('H5S_SIMPLE');
dims = [100 200];
h5_dims = fliplr(dims);
maxdims = [100 H5ML.get_constant_value('H5S_UNLIMITED')];
h5_maxdims = fliplr(maxdims);
H5S.set_extent_simple(space_id1,2,h5_dims,h5_maxdims);
space_id2 = H5S.create('H5S_SIMPLE');
H5S.extent_copy(space_id2,space_id1);
```

See Also

`H5S.create` | `H5S.get_simple_extent_dims` | `H5S.set_extent_simple`

H5S.get_select_bounds

Purpose Bounding box of data space selection

Syntax

```
[start,finish] = H5S.get_select_bounds(space_id)
```

Description

`[start,finish] = H5S.get_select_bounds(space_id)` returns the coordinates of the bounding box containing the current selection. `start` contains the starting coordinates of the bounding box and `finish` contains the coordinates of the diagonally opposite corner.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_start`, `h5_stride`, `h5_count` and `h5_block` parameters assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([10 20]); block = fliplr([20 30]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
start = fliplr([30 40]); block = fliplr([20 30]);
H5S.select_hyperslab(space_id,'H5S_SELECT_OR',start,[],[],block);
[start, finish] = H5S.get_select_bounds(space_id);
matlab_start = fliplr(start);
matlab_finish = fliplr(finish);
```

See Also `H5S.create_simple` | `H5S.select_hyperslab`

H5S.get_select_elem_npoints

Purpose

Number of element points in selection

Syntax

```
numpoints = H5S.get_select_elem_npoints(space_id)
```

Description

`numpoints = H5S.get_select_elem_npoints(space_id)` returns the number of element points in the current data space selection.

Examples

Select the corner points of a data space.

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
coords = [0 0; 0 199; 99 0; 99 199];
coords = fliplr(coords);
coords = coords';
H5S.select_elements(space_id,'H5S_SELECT_SET',coords)
numpoints = H5S.get_select_elem_npoints(space_id);
```

See Also

`H5S.select_elements`

H5S.get_select_elem_pointlist

Purpose

Element points in data space selection

Syntax

```
points = H5S.get_select_elem_pointlist(space_id,startpoint,
   numpoints)
```

Description

`points = H5S.get_select_elem_pointlist(space_id,startpoint,numpoints)` returns the list of element points in the current data space selection. `startpoint` specifies the element point to start with and `numpoints` specifies the total number of points.

`points` is a two-dimensional array of 0-based values specifying the coordinates of the elements. If `m` is the rank of the dataspace, then `points` will have size [`m` x `numpoints`].

Note The ordering of the coordinate points is the same as the HDF5 library C API.

Examples

Determine the first two points in the current selection.

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
coords = [0 0; 0 199; 99 0; 99 199];
coords = fliplr(coords);
coords = coords';
H5S.select_elements(space_id,'H5S_SELECT_SET',coords);
points = H5S.get_select_elem_pointlist(space_id,0,2);
```

Purpose

List of hyperslab blocks

Syntax

```
blocklist = H5S.get_select_hyper_blocklist(space_id,
   startblock,numblocks)
```

Description

`blocklist = H5S.get_select_hyper_blocklist(space_id,startblock,numblocks)` returns a list of the hyperslab blocks currently selected. `space_id` is a dataspace identifier. `startblock` specifies the block to start with and `numblocks` specifies the number of hyperslab blocks to retrieve.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_start`, `h5_stride`, `h5_count` and `h5_block` parameters assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([10 20]); block = fliplr([20 25]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
start = fliplr([20 30]); block = fliplr([20 25]);
H5S.select_hyperslab(space_id,'H5S_SELECT_NOTB',start,[],[],block);
nblocks = H5S.get_select_hyper_nblocks(space_id);
for j = 1:nblocks
    hblocks{j} = H5S.get_select_hyper_blocklist(space_id,j-1,1);
end
```

See Also

`H5S.select_hyperslab` | `H5S.get_select_hyper_nblocks`

H5S.get_select_hyper_nblocks

Purpose

Number of hyperslab blocks

Syntax

```
num_blocks = H5S.get_select_hyper_nblocks(space_id)
```

Description

`num_blocks = H5S.get_select_hyper_nblocks(space_id)` returns the number of hyperslab blocks in the current data space selection.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([10 20]); block = fliplr([20 25]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
start = fliplr([20 30]); block = fliplr([20 25]);
H5S.select_hyperslab(space_id,'H5S_SELECT_NOTB',start,[],[],block);
nblocks = H5S.get_select_hyper_nblocks(space_id);
```

See Also

`H5S.get_select_hyper_blocklist` | `H5S.select_hyperslab`

H5S.get_select_npoints

Purpose

Number of elements in data space selection

Syntax

```
num_points = H5S.get_select_npoints(space_id)
```

Description

`num_points = H5S.get_select_npoints(space_id)` returns the number of elements in the current data space selection.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
op = 'H5S_SELECT_SET';
start = fliplr([10 20]); block = fliplr([20 30]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
n = H5S.get_select_npoints(space_id);
```

See Also

H5S.create_simple | H5S.select_hyperslab

H5S.get_select_type

Purpose

Type of data space selection

Syntax

```
sel_type = H5S.get_select_type(space_id)
```

Description

`sel_type = H5S.get_select_type(space_id)` returns the data space selection type. Valid return values correspond to the following enumerated constants:

- `H5S_SEL_NONE`
- `H5S_SEL_POINTS`
- `H5S_SEL_HYPERSLABS`
- `H5S_SEL_ALL`

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([10 20]); block = fliplr([20 30]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
sel_type = H5S.get_select_type(space_id);
switch(sel_type)
    case H5ML.get_constant_value('H5S_SEL_NONE')
        fprintf('no selection\n');
    case H5ML.get_constant_value('H5S_SEL_POINTS');
        fprintf('point selection\n');
    case H5ML.get_constant_value('H5S_SEL_HYPERSLABS');
        fprintf('hyperslab selection\n');
end
```

See Also

`H5S.select_elements` | `H5S.select_hyperslab` | `H5ML.get_constant_value`

Purpose

Data space size and maximum size

Syntax

```
[numdims,h5_dims,
   h5_maxdims] = H5S.get_simple_extent_dims(space_id)
```

Description

`[numdims,h5_dims,h5_maxdims] = H5S.get_simple_extent_dims(space_id)` returns the number of dimensions in the data space, the size of each dimension, and the maximum size of each dimension.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_dims` and `h5_maxdims` assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g2/dset2.2');
space_id = H5D.get_space(dset_id);
[ndims,h5_dims] = H5S.get_simple_extent_dims(space_id);
matlab_dims = fliplr(h5_dims);
```

H5S.get_simple_extent_ndims

Purpose Data space rank

Syntax

```
output = H5S.get_simple_extent_ndims(space_id)
```

Description

`output = H5S.get_simple_extent_ndims(space_id)` returns the dimensionality (also called the rank) of a data space.

H5S.get_simple_extent_npoints

Purpose

Number of elements in data space

Syntax

```
output = H5S.get_simple_extent_npoints(space_id)
```

Description

`output = H5S.get_simple_extent_npoints(space_id)` returns the number of elements in the data space specified by `space_id`.

H5S.get_simple_extent_type

Purpose

Data space class

Syntax

```
space_type = H5S.get_simple_extent_type(space_id)
```

Description

`space_type = H5S.get_simple_extent_type(space_id)` returns the data space class of the data space specified by `space_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
space_id = H5D.get_space(dset_id);
space_type = H5S.get_simple_extent_type(space_id);
switch(space_type)
    case H5ML.get_constant_value('H5S_SCALAR')
        fprintf('scalar\n');
    case H5ML.get_constant_value('H5S_SIMPLE')
        fprintf('simple\n');
    case H5ML.get_constant_value('H5S_NULL')
        fprintf('none\n');
end
```

See Also

`H5S.create` | `H5D.get_space` | `H5ML.get_constant_value`

H5S.is_simple

Purpose

Determine if data space is simple

Syntax

```
output = H5S.is_simple(space_id)
```

Description

`output = H5S.is_simple(space_id)` returns a positive value if the data space specified by `space_id` is a simple data space, zero if it is not, and a negative value to indicate failure.

Examples

Create a new data space and verify that it is simple.

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
val = H5S.is_simple(space_id);
```

Create a null data space and verify that it is not simple.

```
space_id = H5S.create('H5S_NULL');
val = H5S.is_simple(space_id);
```

See Also

`H5S.create` | `H5S.create_simple`

H5S.offset_simple

Purpose

Set offset of simple data space

Syntax

```
H5S.offset_simple(space_id,offset)
```

Description

`H5S.offset_simple(space_id,offset)` specifies the offset of the simple data space specified by `space_id`. This function allows the same shaped selection to be moved to different locations within a data space without requiring it to be redefined.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_start`, `h5_stride`, `h5_count` and `h5_block` parameters assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([10 20]); block = fliplr([20 30]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
offset = fliplr([3 5]);
H5S.offset_simple(space_id,offset)
[start,finish] = H5S.get_select_bounds(space_id);
start = fliplr(start);
finish = fliplr(finish);
```

See Also

`H5S.get_select_bounds | H5S.select_hyperslab`

Purpose

Select entire extent of data space

Syntax

```
H5S.select_all(space_id)
```

Description

`H5S.select_all(space_id)` selects the entire extent of the data space specified by `space_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
space_id = H5D.get_space(dset_id);
num_points1 = H5S.get_select_npoints(space_id);
H5S.select_none(space_id);
num_points2 = H5S.get_select_npoints(space_id);
H5S.select_all(space_id);
num_points3 = H5S.get_select_npoints(space_id);
```

H5S.select_elements

Purpose

Specify coordinates to include in selection

Syntax

```
H5S.select_elements(space_id,op,h5_coord)
```

Description

`H5S.select_elements(space_id,op,h5_coord)` selects the array elements to be included in the selection for the data space specified by `space_id`. The `op` argument determines how the new selection is to be combined with the previously existing selection for the data space and can be specified by one of the following string values.

`'H5S_SELECT_SET'`

`'H5S_SELECT_APPEND'`

`'H5S_SELECT_PREPEND'`

`h5_coord` is a two-dimensional array of 0-based values specifying the coordinates of the elements being selected. If `m` is the rank of the data space and if n is the number of points, then `h5_coord` should be an `m`-by-`n` array.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_coord` parameter assumes coordinates have C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Select the corner points of a data space. In this case, `h5_coord` should have size 2x4.

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
coords = [0 0; 0 199; 99 0; 99 199];
h5_coords = fliplr(coords);
h5_coords = h5_coords';
```

```
H5S.select_elements(space_id,'H5S_SELECT_SET',h5_coords);
```

See Also

H5S.create_simple | H5S.get_select_elem_npoints | H5S.get_select_elem_pointlist

H5S.select_hyperslab

Purpose

Select hyperslab region

Syntax

```
H5S.select_hyperslab(space_id,op,h5_start,h5_stride,h5_count,
   h5_block)
```

Description

`H5S.select_hyperslab(space_id,op,h5_start,h5_stride,h5_count,h5_block)` selects a hyperslab region to add to the current selected region for the data space specified by `space_id`. The `op` argument determines how the new selection is to be combined with the previously existing selection for the data space. Possible values include: `H5S_SELECT_SET`, `H5S_SELECT_OR`, `H5S_SELECT_AND`, `H5S_SELECT_XOR`, `H5S_SELECT_NOTA`, or `H5S_SELECT_NOTB`.

The `h5_start` array determines the starting coordinates of the hyperslab to select. The `h5_count` array determines how many blocks to select from the data space, in each dimension. The `h5_stride` array specifies how many elements to move in each dimension. The `h5_block` array determines the size of the element block selected from the data space.

If `h5_stride` is specified as `[]`, then a contiguous hyperslab is selected, as if each value in `h5_stride` were set to 1. If `h5_count` is specified as `[]`, the number of blocks selected along each dimension defaults to 1. If `h5_block` is specified as `[]`, then the block size defaults to a single element in each dimension, as if each value in the block array were set to 1.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_start`, `h5_stride`, `h5_count` and `h5_block` parameters assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
```

```
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([10 20]); block = fliplr([20 30]);
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],[],block);
```

See Also

`H5S.create_simple`

H5S.select_none

Purpose

Reset selection region to include no elements

Syntax

```
H5S.select_none(space_id)
```

Description

`H5S.select_none(space_id)` resets the selection region for the data space `space_id` to include no elements.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
space_id = H5D.get_space(dset_id);
num_points1 = H5S.get_select_npoints(space_id);
H5S.select_none(space_id);
num_points2 = H5S.get_select_npoints(space_id);
```

Purpose

Determine validity of selection

Syntax

```
output = H5S.select_valid(space_id)
```

Description

`output = H5S.select_valid(space_id)` returns a positive value if the selection of the data space `space_id` is within the extent of that data space, and zero if it is not. A negative value indicates failure.

Examples

```
dims = [100 200];
h5_dims = fliplr(dims);
space_id = H5S.create_simple(2,h5_dims,h5_dims);
start = fliplr([90 190]); count = [11 11];
H5S.select_hyperslab(space_id,'H5S_SELECT_SET',start,[],count,[]);
valid = H5S.select_valid(space_id);
```

See Also

`H5S.create_simple` | `H5S.select_hyperslab`

H5S.set_extent_none

Purpose

Remove extent from data space

Syntax

```
H5S.set_extent_none(space_id)
```

Description

`H5S.set_extent_none(space_id)` removes the extent from a data space and sets the type to `H5S_NO_CLASS`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer2D');
space_id = H5D.get_space(dset_id);
H5S.set_extent_none(space_id);
extent_type = H5S.get_simple_extent_type(space_id);
switch(extent_type)
    case H5ML.get_constant_value('H5S_SCALAR')
        fprintf('scalar\n');
    case H5ML.get_constant_value('H5S_SIMPLE')
        fprintf('simple\n');
    case H5ML.get_constant_value('H5S_NO_CLASS')
        fprintf('no class\n');
end
```

See Also

`H5S.get_simple_extent_dims`

Purpose

Set size of data space

Syntax

```
H5S.set_extent_simple(space_id,rank,h5_dims,h5_maxdims)
```

Description

`H5S.set_extent_simple(space_id,rank,h5_dims,h5_maxdims)` sets the size of the data space identified by `space_id`. The `rank` argument is the number of dimensions used in the data space. `h5_dims` is an array specifying the size of each dimension of the dataset. `h5_maxdims` is an array specifying the upper limit on the size of each dimension.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_dims` and `h5_maxdims` parameters assume C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
space_id = H5S.create('H5S_SIMPLE');
dims = [100 200];
h5_dims = fliplr(dims);
maxdims = [100 H5ML.get_constant_value('H5S_UNLIMITED')];
h5_maxdims = fliplr(maxdims);
H5S.set_extent_simple(space_id,2,h5_dims, h5_maxdims);
```

See Also

`H5S.create` | `H5S.get_simple_extent_dims` | `H5ML.get_constant_value`

H5T.close

Purpose

Close data type

Syntax

```
H5T.close(type_id)
```

Description

`H5T.close(type_id)` releases the data type specified by `type_id`.

See Also

`H5A.get_type | H5D.get_type`

Purpose

Commit transient data type

Syntax

```
H5T.commit(loc_id,name,type_id)
H5T.commit(loc_id,name,type_id,lcpl_id,tcpl_id,tapl_id)
```

Description

`H5T.commit(loc_id,name,type_id)` commits a transient data type to a file, creating a new named data type. `loc_id` is a file or group identifier. `name` is the name of the data type and `type_id` is the data type id. This interface corresponds to the 1.6.x version of `H5Tcommit`.

`H5T.commit(loc_id,name,type_id,lcpl_id,tcpl_id,tapl_id)` commits a transient data type to a file, creating a new named data type. `loc_id` is a file or group identifier. `name` is the name of the data type and `type_id` is the data type id. `lcpl_id`, `tcpl_id`, and `tapl_id` are link creation, data type creation, and data type access property list identifiers. This interface corresponds to the 1.8.x version of `H5Tcommit`.

Examples

Create a named variable-length data type.

```
plist_id = 'H5P_DEFAULT';
fid = H5F.create('myfile.h5','H5F_ACC_TRUNC',plist_id,plist_id);
base_type_id = H5T.copy('H5T_NATIVE_DOUBLE');
vlen_type_id = H5T.vlen_create(base_type_id);
H5T.commit(fid,'MyVlen',vlen_type_id);
H5T.close(vlen_type_id);
H5T.close(base_type_id);
H5F.close(fid);
```

See Also

`H5T.close` | `H5T.committed`

H5T.committed

Purpose

Determine if data type is committed

Syntax

```
output = H5T.committed(type_id)
```

Description

`output = H5T.committed(type_id)` returns a positive value to indicate that the data type has been committed, and zero to indicate that it has not. A negative value indicates failure.

Examples

```
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
is_committed = H5T.committed(type_id);
```

See Also

`H5T.commit`

H5T.copy

Purpose

Copy data type

Syntax

```
output_type_id = H5T.copy(type_id)
```

Description

`output_type_id = H5T.copy(type_id)` copies the existing data type identifier, a dataset identifier specified by `type_id`, or a predefined data type such as `'H5T_NATIVE_DOUBLE'`. `output_type_id` is a data type identifier.

Examples

```
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
type_size = H5T.get_size(type_id);
```

See Also

`H5T.get_size`

H5T.create

Purpose

Create new data type

Syntax

```
output = H5T.create(class_id,size)
```

Description

`output = H5T.create(class_id,size)` creates a new data type of the class specified by `class_id`, with the number of bytes specified by `size`. The `output` argument is a data type identifier.

Examples

Create a signed 32-bit enumerated data type.

```
type_id = H5T.create('H5T_ENUM',4);
H5T.set_order(type_id,'H5T_ORDER_LE');
H5T.set_sign(type_id,'H5T_SGN_2');
H5T.enum_insert(type_id,'black',0);
H5T.enum_insert(type_id,'white',1);
```

See Also

`H5T.set_order | H5T.set_sign`

Purpose

Determine of data type contains specific class

Syntax

```
output = H5T.detect_class(type_id,class_id)
```

Description

`output = H5T.detect_class(type_id,class_id)` returns a positive value if the data type specified in `type_id` contains any data types of the data type class specified in `class_id`, or zero to indicate that it does not. A negative value indicates failure.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/vlen');
type_id = H5D.get_type(dset_id);
has_double = H5T.detect_class(type_id,'H5T_FLOAT');
```

See Also

`H5D.get_type`

H5T.equal

Purpose

Determine equality of data types

Syntax

```
output = H5T.equal(type1_id,type2_id)
```

Description

`output = H5T.equal(type1_id,type2_id)` returns a positive number if the data type identifiers refer to the same data type, and zero to indicate that they do not. A negative value indicates failure. Either of the input values could be a string corresponding to an HDF5 data type.

Examples

Determine if the data type of a dataset is a 32-bit little endian integer.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer2D');
dtype_id = H5D.get_type(dset_id);
if H5T.equal(dtype_id,'H5T_STD_I32LE')
    fprintf('32-bit little endian integer\n');
end
```

See Also

`H5D.get_type`

Purpose Data type class identifier

Syntax

```
class_id = H5T.get_class(type_id)
```

Description

`class_id = H5T.get_class(type_id)` returns the data type class identifier of the data type specified by `type_id`.

Valid class identifiers include:

- H5T_INTEGER
- H5T_FLOAT
- H5T_STRING
- H5T_BITFIELD
- H5T_OPAQUE
- H5T_COMPOUND
- H5T_ENUM
- H5T_VLEN
- H5T_ARRAY

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/enum');
type_id = H5D.get_type(dset_id);
class_id = H5T.get_class(type_id);
switch(class_id)
    case H5ML.get_constant_value('H5T_INTEGER')
        fprintf('Integer\n');
    case H5ML.get_constant_value('H5T_FLOAT')
        fprintf('Floating point\n');
    case H5ML.get_constant_value('H5T_STRING')
        fprintf('String\n');
    case H5ML.get_constant_value('H5T_BITFIELD')
        fprintf('Bitfield\n');
    case H5ML.get_constant_value('H5T_OPAQUE')
```

```
            fprintf('Opaque\n');
        case H5ML.get_constant_value('H5T_COMPOUND')
            fprintf('Compound'\n');
        case H5ML.get_constant_value('H5T_ENUM')
            fprintf('Enumerated\n');
        case H5ML.get_constant_value('H5T_VLEN')
            fprintf('Variable length\n');
        case H5ML.get_constant_value('H5T_ARRAY')
            fprintf('Array\n');
    end
```

See Also

`H5ML.get_constant_value`

H5T.get_create_plist

Purpose

Copy of data type creation property list

Syntax

```
plist_id = H5T.get_create_plist(datatype_id)
```

Description

`plist_id = H5T.get_create_plist(datatype_id)` returns a property list identifier for the data type creation property list associated with the data type specified by `datatype_id`.

See Also

`H5D.get_create_plist` | `H5F.get_create_plist`

H5T.get_native_type

Purpose Native data type of dataset data type

Syntax

```
output = H5T.get_native_type(type_id,direction)
```

Description

`output = H5T.get_native_type(type_id,direction)` returns the equivalent native data type for the dataset data type specified in `type_id`. The `direction` argument indicates the order in which the library searches for a native data type match and must be either `'H5T_DIR_ASCEND'` or `'H5T_DIR_DESCEND'`.

Purpose

Size of data type in bytes

Syntax

```
type_size = H5T.get_size(type_id)
```

Description

`type_size = H5T.get_size(type_id)` returns the size of a data type in bytes. `type_id` is a data type identifier.

Examples

Determine the size of the data type for a specific dataset.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/bitfield2D');
type_id = H5D.get_type(dset_id);
type_size = H5T.get_size(type_id);
```

See Also

`H5T.set_size | H5D.get_type`

H5T.get_super

Purpose

Base data type

Syntax

```
super_type_id = H5T.get_super(type_id)
```

Description

`super_type_id = H5T.get_super(type_id)` returns the base data type from which the data type type specified by `type_id` is derived.

Examples

Retrieve the base data type for an enumerated dataset.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/enum');
dtype_id = H5D.get_type(dset_id);
super_type_id = H5T.get_super(dtype_id);
```

Purpose

Lock data type

Syntax

```
H5T.lock(type_id)
```

Description

`H5T.lock(type_id)` locks the data type specified by `type_id`, making it read-only and non-destructible.

H5T.open

Purpose

Open named data type

Syntax

```
type_id = H5T.open(loc_id,name)
```

Description

`type_id = H5T.open(loc_id,name)` opens a named data type at the location specified by `loc_id` and returns an identifier for the data type. `loc_id` is either a file or group identifier.

This function corresponds to the `H5Topen1` function in the HDF5 library C API.

See Also

`H5T.close` | `H5A.open` | `H5D.open` | `H5G.open` | `H5O.open`

Purpose

Create array data type object

Syntax

```
array_type_id = H5T.array_create(base_id,h5_dims)
array_type_id = H5T.array_create(base_id,rank,h5_dims,perms)
```

Description

`array_type_id = H5T.array_create(base_id,h5_dims)` creates a new array data type object. This interface corresponds to the 1.8 library version of `H5Tarray_create`.

`array_type_id = H5T.array_create(base_id,rank,h5_dims,perms)` creates a new array data type object. This interface corresponds to the 1.6 library version of `H5Tarray_create`. The `perms` parameter is not used at this time and can be omitted.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. The `h5_dims` parameter assumes C-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

Create a 100-by-200 double precision array data type.

```
base_type_id = H5T.copy('H5T_NATIVE_DOUBLE');
dims = [100 200];
h5_dims = fliplr(dims);
array_type = H5T.array_create(base_type_id,h5_dims);
```

See Also

`H5T.get_array_dims` | `H5T.get_array_ndims`

H5T.get_array_dims

Purpose Array dimension extents

Syntax

```
dimsizes = H5T.get_array_dims(type_id)
[ndims,dimsizes,perm] = H5T.get_array_dims(type_id)
```

Description

`dimsizes = H5T.get_array_dims(type_id)` returns the sizes of the dimensions and the dimension permutations of the specified array data type object. This interface corresponds to the 1.8 version of `H5Tget_array_dims`.

`[ndims,dimsizes,perm] = H5T.get_array_dims(type_id)` corresponds to the 1.6 version of the interface. It is strongly deprecated.

Note The HDF5 library uses C-style ordering for multidimensional arrays, while MATLAB uses FORTRAN-style ordering. Please consult "Using the MATLAB Low-Level HDF5 Functions" in the MATLAB documentation for more information.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/array2D');
type_id = H5D.get_type(dset_id);
h5_dims = H5T.get_array_dims(type_id);
dims = fliplr(h5_dims);
```

See Also `H5T.array_create | H5T.get_array_ndims`

H5T.get_array_ndims

Purpose

Rank of array data type

Syntax

```
output = H5T.get_array_ndims(type_id)
```

Description

`output = H5T.get_array_ndims(type_id)` returns the rank, the number of dimensions, of an array data type object.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/array2D');
type_id = H5D.get_type(dset_id);
ndims = H5T.get_array_ndims(type_id);
```

See Also

`H5T.get_array_dims`

H5T.get_cset

Purpose

Character set of string data type

Syntax

```
cset = H5T.get_cset(type_id)
```

Description

`cset = H5T.get_cset(type_id)` returns the character set type of the data type specified by `type_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/string');
type_id = H5D.get_type(dset_id);
cset = H5T.get_cset(type_id);
switch(cset)
  case H5ML.get_constant_value('H5T_CSET_ASCII')
      fprintf('ASCII\n');
  case H5ML.get_constant_value('H5T_CSET_UTF8')
      fprintf('UTF-8\n');
end
```

See Also

`H5T.set_cset`

Purpose

Exponent bias of floating-point type

Syntax

```
output = H5T.get_ebias(type_id)
```

Description

`output = H5T.get_ebias(type_id)` returns the exponent bias of a floating-point type. `type_id` is data type identifier.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
type_id = H5D.get_type(dset_id);
ebias = H5T.get_ebias(type_id);
```

See Also

`H5T.set_ebias`

H5T.get_fields

Purpose

Floating-point data type bit field information

Syntax

```
[spos,epos,esize,mpos,msize] = H5T.get_fields(type_id)
```

Description

`[spos,epos,esize,mpos,msize] = H5T.get_fields(type_id)` returns information about the locations of the various bit fields of a floating point data type. `type_id` is a data type identifier. `spos` is the floating-point sign bit. `epos` is the exponent bit-position. `esize` is the size of the exponent in bits. `mpos` is the mantissa bit-position. `msize` is the size of the mantissa in bits.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
type_id = H5D.get_type(dset_id);
[spos, epos, esize, mpos, msize] = H5T.get_fields(type_id);
```

Purpose

Internal padding type for floating-point data types

Syntax

```
pad_type = H5T.get_inpad(type_id)
```

Description

`pad_type = H5T.get_inpad(type_id)` returns the internal padding type for unused bits in floating-point data types. `type_id` is a data type identifier. `pad_type` can be `H5T_PAD_ZERO`, `H5T_PAD_ONE`, or `H5T_PAD_BACKGROUND`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
type_id = H5D.get_type(dset_id);
pad_type = H5T.get_inpad(type_id);
switch(pad_type)
    case H5ML.get_constant_value('H5T_PAD_ZERO')
        fprintf('pad zero\n');
    case H5ML.get_constant_value('H5T_PAD_ONE');
        fprintf('pad one\n');
    case H5ML.get_constant_value('H5T_PAD_BACKGROUND')
        fprintf('pad background\n');
end
```

See Also

`H5T.set_inpad`

H5T.get_norm

Purpose

Mantissa normalization type

Syntax

```
norm_type = H5T.get_norm(type_id)
```

Description

`norm_type = H5T.get_norm(type_id)` returns the mantissa normalization of a floating-point data type. `type_id` is a data type identifier. `norm_type` can be `H5T_NORM_IMPLIED`, `H5T_NORM_MSBSET`, or `H5T_NORM_NONE`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
type_id = H5D.get_type(dset_id);
norm_type = H5T.get_norm(type_id);
switch(norm_type)
    case H5ML.get_constant_value('H5T_NORM_IMPLIED')
        fprintf('MSB of mantissa is not stored, always 1\n');
    case H5ML.get_constant_value('H5T_NORM_MSBSET');
        fprintf('MSB of mantissa is always 1\n');
    case H5ML.get_constant_value('H5T_NORM_NONE')
        fprintf('mantissa is not normalized\n');
end
```

See Also

`H5T.set_norm`

H5T.get_offset

Purpose Bit offset of first significant bit

Syntax `offset = H5T.get_offset(type_id)`

Description `offset = H5T.get_offset(type_id)` returns the offset of the first significant bit. `type_id` is a data type identifier.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/float');
type_id = H5D.get_type(dset_id);
offset = H5T.get_offset(type_id);
```

See Also `H5T.set_offset`

H5T.get_order

Purpose

Byte order of atomic data type

Syntax

```
output = H5T.get_order(type_id)
```

Description

`output = H5T.get_order(type_id)` returns the byte order of an atomic data type. `type_id` is a data type identifier. Possible return values are the constant values corresponding to the following strings:

- `'H5T_ORDER_LE'`
- `'H5T_ORDER_BE'`
- `'H5T_ORDER_VAX'`

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g2/dset2.2');
type_id = H5D.get_type(dset_id);
switch(H5T.get_order(type_id))
    case H5ML.get_constant_value('H5T_ORDER_LE')
        fprintf('little endian\n');
    case H5ML.get_constant_value('H5T_ORDER_BE')
        fprintf('big endian\n');
    case H5ML.get_constant_value('H5T_ORDER_VAX')
        fprintf('vax\n');
end
```

See Also

`H5T.set_order` | `H5ML.get_constant_value`

Purpose

Padding type of least and most-significant bits

Syntax

```
[lsb,msb] = H5T.get_pad(type_id)
```

Description

`[lsb,msb] = H5T.get_pad(type_id)` returns the padding type of the least and most-significant bit padding. `type_id` is a data type identifier. `lsb` is the least-significant bit padding type. `msb` is the most-significant bit padding type. Values for `lsb` and `msb` can be `H5T_PAD_ZERO`, `H5T_PAD_ONE`, or `H5T_PAD_BACKGROUND`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
type_id = H5D.get_type(dset_id);
[lsb,msb] = H5T.get_pad(type_id);
switch(lsb)
    case H5ML.get_constant_value('H5T_PAD_ZERO')
        fprintf('lsb pad type is zeros\n');
    case H5ML.get_constant_value('H5T_PAD_ONE');
        fprintf('lsb pad type is ones\n');
    case H5ML.get_constant_value('H5T_PAD_BACKGROUND')
        fprintf('lsb pad type is background\n');
end
switch(msb)
    case H5ML.get_constant_value('H5T_PAD_ZERO')
        fprintf('msb pad type is zeros\n');
    case H5ML.get_constant_value('H5T_PAD_ONE');
        fprintf('msb pad type is ones\n');
    case H5ML.get_constant_value('H5T_PAD_BACKGROUND')
        fprintf('msb pad type is background\n');
end
```

See Also

`H5T.set_pad`

H5T.get_precision

Purpose Precision of atomic data type

Syntax

```
output = H5T.get_precision(type_id)
```

Description `output = H5T.get_precision(type_id)` returns the precision of an atomic data type. `type_id` is a data type identifier.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
type_id = H5D.get_type(dset_id);
numbits = H5T.get_precision(type_id);
```

See Also `H5T.set_precision`

Purpose

Sign type for integer data type

Syntax

```
sign_type = H5T.get_sign(type_id)
```

Description

`sign_type = H5T.get_sign(type_id)` returns the sign type for an integer type. `type_id` is a data type identifier. Valid types are: `H5T_SGN_NONE` or `H5T_SGN_2`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/integer');
type_id = H5D.get_type(dset_id);
sign_type = H5T.get_sign(type_id);
switch(sign_type)
    case H5ML.get_constant_value('H5T_SGN_NONE')
        fprintf('Unsigned integer type.\n');
    case H5ML.get_constant_value('H5T_SGN_2');
        fprintf('Signed integer type.\n');
end
```

See Also

`H5T.set_sign`

H5T.get_strpad

Purpose

Storage mechanism for string data type

Syntax

```
output = H5T.get_strpad(type_id)
```

Description

`output = H5T.get_strpad(type_id)` returns the storage mechanism (padding type) for a string data type. Possible values are:

H5T_STR_NULLPAD	Pad with zeros
H5T_STR_NULLTERM	Null-terminate
H5T_STR_SPACEPAD	Pad with spaces

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/string');
type_id = H5D.get_type(dset_id);
padding = H5T.get_strpad(type_id);
switch(padding)
    case H5ML.get_constant_value('H5T_STR_NULLTERM')
        fprintf('null-terminated\n');
    case H5ML.get_constant_value('H5T_STR_NULLPAD')
        fprintf('padded with zeros\n');
    case H5ML.get_constant_value('H5T_STR_SPACEPAD')
        fprintf('padded with spaces\n');
end
```

See Also

H5T.set_strpad

Purpose

Set character dataset for string data type

Syntax

```
H5T.set_cset(type_id, cset)
```

Description

`H5T.set_cset(type_id, cset)` sets the character encoding used to create strings. The only valid type is `H5T_CSET_ASCII`.

Examples

```
type_id = H5T.copy('H5T_C_S1');
H5T.set_size(type_id,10);
encoding = H5ML.get_constant_value('H5T_CSET_ASCII');
H5T.set_cset(type_id,encoding);
```

See Also

`H5T.get_cset`

H5T.set_ebias

Purpose

Set exponent bias of floating-point data type

Syntax

```
H5T.set_ebias(type_id,ebias)
```

Description

`H5T.set_ebias(type_id,ebias)` sets the exponent bias of a floating-point type. `type_id` is a data type identifier. `ebias` is an exponent bias value.

Examples

```
type_id = H5T.copy('H5T_NATIVE_FLOAT');
H5T.set_size(type_id,32);
H5T.set_ebias(type_id,99);
```

See Also

`H5T.get_ebias`

Purpose Set sizes and locations of floating-point bit fields

Syntax

```
H5T.set_fields(type_id,spos,epos,esize,mpos,msize)
```

Description `H5T.set_fields(type_id,spos,epos,esize,mpos,msize)` sets the locations and sizes of the various floating-point bit fields. `spos` is the sign position. `epos` is the exponent in bits. `esize` is the size of exponent in bits. `mpos` is the mantissa bit position. `msize` is the size of the mantissa in bits.

Examples

```
type_id = H5T.copy('H5T_NATIVE_DOUBLE');
H5T.set_fields(type_id,30,24,6,0,2);
```

See Also `H5T.get_fields`

H5T.set_inpad

Purpose Specify how unused internal bits are to be filled

Syntax `H5T.set_inpad(type_id,pad)`

Description `H5T.set_inpad(type_id,pad)` sets how unused internal bits of a floating point type are filled. `type_id` is the identifier of the data type. `inpad` specifies how to fill the bits: `H5T_PAD_ZERO`, `H5T_PAD_ONE`, or `H5T_PAD_BACKGROUND` (leave background alone).

Examples

```
type_id = H5T.copy('H5T_NATIVE_FLOAT');
pad_type = H5ML.get_constant_value('H5T_PAD_ZERO');
H5T.set_inpad(type_id,pad_type);
```

See Also `H5T.get_inpad`

H5T.set_norm

Purpose

Set mantissa normalization of floating-point data type

Syntax

```
H5T.set_norm(type_id,norm)
```

Description

`H5T.set_norm(type_id,norm)` sets the mantissa normalization of a floating-point data type. Valid normalization types are: `H5T_NORM_IMPLIED`, `H5T_NORM_MSBSET`, or `H5T_NORM_NONE`.

Examples

```
type_id = H5T.copy('H5T_NATIVE_FLOAT');
norm_type = H5ML.get_constant_value('H5T_NORM_MSBSET');
H5T.set_norm(type_id,norm_type);
```

See Also

`H5T.get_norm`

H5T.set_offset

Purpose

Set bit offset of first significant bit

Syntax

```
H5T.set_offset(type_id,offset)
```

Description

`H5T.set_offset(type_id,offset)` sets the bit offset of the first significant bit. `type_id` is the identifier of the data type. `offset` specifies the number of bits of padding that appear.

Examples

```
type_id = H5T.copy('H5T_NATIVE_INT');
H5T.set_offset(type_id,16);
```

See Also

`H5T.get_offset`

Purpose

Set byte ordering of atomic data type

Syntax

```
H5T.set_order(type_id,type_order)
```

Description

`H5T.set_order(type_id,type_order)` sets the byte ordering of an atomic data type. `type_order` can be one of the following values:

- `H5T_ORDER_LE`
- `H5T_ORDER_BE`
- `H5T_ORDER_VAX`

Examples

Create a big endian 32-bit integer type.

```
type_id = H5T.copy('H5T_NATIVE_INT');
order = H5ML.get_constant_value('H5T_ORDER_BE');
H5T.set_order(type_id,order);
```

See Also

`H5T.get_order` | `H5ML.get_constant_value`

H5T.set_pad

Purpose

Set padding type for least and most significant bits

Syntax

```
H5T.set_pad(type_id,lsb,msb)
```

Description

`H5T.set_pad(type_id,lsb,msb)` sets the padding type for the least and most-significant bits. `type_id` is the identifier of the data type. `lsb` specifies the padding type for least-significant bits; `msb` for most-significant bits. Valid padding types are `H5T_PAD_ZERO`, `H5T_PAD_ONE`, or `H5T_PAD_BACKGROUND` (leave background alone).

Examples

```
type_id = H5T.copy('H5T_NATIVE_INT');
lsb = H5ML.get_constant_value('H5T_PAD_ONE');
msb = H5ML.get_constant_value('H5T_PAD_ZERO');
H5T.set_pad(type_id,lsb,msb);
```

See Also

`H5T.get_pad`

Purpose

Set precision of atomic data type

Syntax

```
H5T.set_precision(type_id,prec)
```

Description

`H5T.set_precision(type_id,prec)` sets the precision of an atomic data type. `type_id` is a data type identifier. `prec` specifies the number of bits of precision for the data type.

H5T.set_sign

Purpose

Set sign property for integer data type

Syntax

```
H5T.set_sign(type_id,sign)
```

Description

`H5T.set_sign(type_id,sign)` sets the sign property for an integer type. `type_id` is a data type identifier. `sign` specifies the sign type. Valid values are `H5T_SGN_NONE` or `H5T_SGN_2`.

Examples

```
type_id = H5T.copy('H5T_NATIVE_LONG');
sgn = H5ML.get_constant_value('H5T_SGN_NONE');
H5T.set_sign(type_id,sgn);
```

See Also

`H5T.get_sign`

Purpose

Set size of data type in bytes

Syntax

```
H5T.set_size(type_id,type_size)
```

Description

`H5T.set_size(type_id,type_size)` sets the total size in bytes for the data type specified by `type_id`. The string `'H5T_VARIABLE'` can also be used if a variable length string is desired.

Examples

Create a variable length string with null termination.

```
type_id = H5T.copy('H5T_C_S1');
H5T.set_size(type_id,'H5T_VARIABLE');
H5T.set_strpad(type_id,'H5T_STR_NULLTERM');
```

See Also

`H5T.get_size`

H5T.set_strpad

Purpose

Set storage mechanism for string data type

Syntax

```
H5T.set_strpad(type_id,storage_type)
```

Description

`H5T.set_strpad(type_id,storage_type)` defines the storage mechanism for the string data type identified by `type_id`. The storage type may be one of the following values.

'H5T_STR_NULLTERM'	Null terminated
'H5T_STR_NULLPAD'	Padded with zeros
'H5T_STR_SPACEPAD'	Padded with spaces

Examples

Create a ten-character string data type with space padding.

```
type_id = H5T.copy('H5T_C_S1');
H5T.set_size(type_id,10);
H5T.set_strpad(type_id,'H5T_STR_SPACEPAD');
```

See Also

`H5T.get_strpad`

Purpose Data type class for compound data type member

Syntax

```
output = H5T.get_member_class(type_id,membno)
```

Description `output = H5T.get_member_class(type_id,membno)` returns the data type class of the compound data type member specified by `membno`. The `type_id` argument is the data type identifier of a compound object.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/compound');
type_id = H5D.get_type(dset_id);
member_name = H5T.get_member_name(type_id,0);
member_class = H5T.get_member_class(type_id,0);
```

See Also `H5T.get_member_name`

H5T.get_member_index

Purpose

Index of compound or enumeration type member

Syntax

```
idx = H5T.get_member_index(type_id,name)
```

Description

`idx = H5T.get_member_index(type_id,name)` returns the index of a field of a compound data type or an element of an enumeration data type. `type_id` is a data type identifier and name is a text string that identifies the target field or element.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/compound');
type_id = H5D.get_type(dset_id);
idx = H5T.get_member_index(type_id,'b');
```

See Also

`H5T.get_member_name`

H5T.get_member_name

Purpose Name of compound or enumeration type member

Syntax

```
name = H5T.get_member_name(type_id,membno)
```

Description `name = H5T.get_member_name(type_id,membno)` returns the name of a field of a compound data type or an element of an enumeration data type. `type_id` is a data type identifier. `membno` is a zero-based index of the field or element whose name is to be retrieved.

Examples Determine the name of the first field of a compound dataset.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/compound');
dtype_id = H5D.get_type(dset_id);
member_name = H5T.get_member_name(dtype_id,0);
```

See Also `H5T.get_member_index`

H5T.get_member_offset

Purpose Offset of field of compound data type

Syntax

```
output = H5T.get_member_offset(type_id,membno)
```

Description

`output = H5T.get_member_offset(type_id,membno)` returns the byte offset of the field specified by `membno` in the compound data type specified by `type_id`. Note that zero (0) is a valid offset.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/compound');
type_id = H5D.get_type(dset_id);
idx = H5T.get_member_offset(type_id,1);
```

See Also `H5T.get_member_name`

H5T.get_member_type

Purpose Data type of specified member

Syntax

```
type_id = H5T.get_member_type(type_id,membno)
```

Description

`type_id = H5T.get_member_type(type_id,membno)` returns the data type of the member specified by `membno` in the data type specified by `type_id`.

Examples

Get the size of the data type of the first member of a compound data type.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/compound');
compound_type_id = H5D.get_type(dset_id);
member_type_id = H5T.get_member_type(compound_type_id,0);
type_size = H5T.get_size(member_type_id);
```

See Also `H5D.get_type`

H5T.get_nmembers

Purpose Number of elements in enumeration type

Syntax

```
output = H5T.get_nmembers(type_id)
```

Description `output = H5T.get_nmembers(type_id)` retrieves the number of fields in a compound data type or the number of members of an enumeration data type. `type_id` is a data type identifier.

Examples Determine the number of fields in a compound dataset.

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/compound');
dtype_id = H5D.get_type(dset_id);
nmembers = H5T.get_nmembers(dtype_id);
```

Purpose Add member to compound data type

Syntax

```
H5T.insert(type_id,name,offset,member_datatype)
```

Description `H5T.insert(type_id,name,offset,member_datatype)` adds another member to the compound data type specified by `type_id`. The `name` argument is a text string that specifies the name of the new member, which must be unique in the compound data type. `offset` specifies where you want to insert the new member and `member_datatype` specifies the data type identifier of the new member.

Examples

```
type_id = H5T.create('H5T_COMPOUND',16);
H5T.insert(type_id,'first',0,'H5T_NATIVE_DOUBLE');
H5T.insert(type_id,'second',8,'H5T_NATIVE_INT');
H5T.insert(type_id,'third',12,'H5T_NATIVE_UINT');
```

See Also `H5T.create`

H5T.pack

Purpose

Recursively remove padding from compound data type

Syntax

```
H5T.pack(type_id)
```

Description

`H5T.pack(type_id)` recursively removes padding from within a compound data type to make it more efficient (space-wise) to store that data. `type_id` is a data type identifier.

Purpose

Create new enumeration data type

Syntax

```
output = H5T.enum_create(parent_id)
```

Description

`output = H5T.enum_create(parent_id)` creates a new enumeration data type based on the specified base data type, `parent_id`, which must be an integer type. `output` is a data type identifier for the new enumeration data type.

Examples

```
parent_id = H5T.copy('H5T_NATIVE_UINT');
type_id = H5T.enum_create(parent_id);
H5T.enum_insert(type_id,'red',1);
H5T.enum_insert(type_id,'green',2);
H5T.enum_insert(type_id,'blue',3);
H5T.close(type_id);
H5T.close(parent_id);
```

See Also

`H5T.enum_insert`

H5T.enum_insert

Purpose Insert enumeration data type member

Syntax

```
H5T.enum_insert(type_id,name,value)
```

Description `H5T.enum_insert(type_id,name,value)` inserts a new enumeration data type member into the enumeration data type specified by `type_id`. The `name` argument is a text string that specifies the name of the new member of the enumeration and `value` is the value of the member.

Examples

```
parent_id = H5T.copy('H5T_NATIVE_UINT');
type_id = H5T.enum_create(parent_id);
H5T.enum_insert(type_id,'red',1);
H5T.enum_insert(type_id,'green',2);
H5T.enum_insert(type_id,'blue',3);
H5T.close(type_id);
H5T.close(parent_id);
```

See Also `H5T.enum_create`

Purpose Name of enumeration data type member

Syntax

```
name = H5T.enum_nameof(type_id,value)
```

Description `name = H5T.enum_nameof(type_id,value)` returns the symbol name corresponding to a member of an enumeration data type. `type_id` specifies the enumeration data type. `value` identifies the member of the enumeration.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/enum');
type_id = H5D.get_type(dset_id);
name0 = H5T.enum_nameof(type_id,int32(0));
name1 = H5T.enum_nameof(type_id,int32(1));
```

See Also `H5T.enum_valueof`

H5T.enum_valueof

Purpose

Value of enumeration data type member

Syntax

```
value = H5T.enum_valueof(type_id,member_name)
```

Description

`value = H5T.enum_valueof(type_id,member_name)` returns the value corresponding to a specified member of an enumeration data type. `type_id` specifies the enumeration data type and `member_name` specifies the member.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/enum');
type_id = H5D.get_type(dset_id);
num_members = H5T.get_nmembers(type_id);
for j = 1:num_members
    member_name{j} = H5T.get_member_name(type_id,j-1);
    member_value(j) = H5T.enum_valueof(type_id,member_name{j});
end
```

See Also

`H5T.get_member_name` | `H5T.get_nmembers`

Purpose

Value of enumeration data type member

Syntax

```
value = H5T.get_member_value(type_id,membno)
```

Description

`value = H5T.get_member_value(type_id,membno)` returns the value of the enumeration data type member specified by `membno`. The `type_id` argument is the data type identifier for the enumeration data type.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/enum');
type_id = H5D.get_type(dset_id);
num_members = H5T.get_nmembers(type_id);
for j = 1:num_members
    member_name{j} = H5T.get_member_name(type_id,j-1);
    member_value(j) = H5T.get_member_value(type_id,j-1);
end
```

See Also

`H5T.get_member_name` | `H5T.get_nmembers`

H5T.get_tag

Purpose Tag associated with opaque data type

Syntax

```
tag = H5T.get_tag(type_id)
```

Description `tag = H5T.get_tag(type_id)` returns the tag associated with the opaque data type specified by `type_id`.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/opaque');
dtype_id = H5D.get_type(dset_id);
tag = H5T.get_tag(dtype_id);
```

See Also `H5T.set_tag`

H5T.set_tag

Purpose

Tag opaque data type with description

Syntax

```
H5T.set_tag(type_id,tag)
```

Description

`H5T.set_tag(type_id,tag)` tags the opaque data type specified by `type_id`, with the descriptive ASCII string identifier, `tag`.

Examples

Create an opaque data type with a length of 4 bytes and a particular tag.

```
type_id = H5T.create('H5T_OPAQUE',4);
H5T.set_tag(type_id,'Created by MATLAB.');
```

See Also

`H5T.create` | `H5T.get_tag`

H5T.is_variable_str

Purpose

Determine if data type is variable-length string

Syntax

```
output = H5T.is_variable_str(type_id)
```

Description

`output = H5T.is_variable_str(type_id)` returns a positive value if the data type specified by `type_id` is a variable-length string and zero if it is not. A negative value indicates failure.

Examples

```
fid = H5F.open('example.h5');
dset_id = H5D.open(fid,'/g3/VLstring2D');
type_id = H5D.get_type(dset_id);
if H5T.is_variable_str(type_id) > 0
    fprintf('variable length string\n');
end
```

See Also

`H5T.vlen_create | H5T.get_size | H5D.get_type`

Purpose

Create new variable-length data type

Syntax

```
vlen_type_id = H5T.vlen_create(base_id)
```

Description

`vlen_type_id = H5T.vlen_create(base_id)` creates a new variable-length (VL) data type. `base_id` specifies the base type of the data type to create.

Examples

Create a variable length data type for 64-bit floating-point numbers.

```
base_type_id = H5T.copy('H5T_NATIVE_DOUBLE');
vlen_type_id = H5T.vlen_create(base_type_id);
```

See Also

`H5T.is_variable_str`

H5Z.filter_avail

Purpose Determine if filter is available

Syntax

```
output = H5Z.filter_avail(filter_id)
```

Description

`output = H5Z.filter_avail(filter_id)` determines whether the filter specified by the filter identifier is available to the application. `filter_id` can be specified by one of the following strings or its numeric equivalent.

- 'H5Z_FILTER_DEFLATE'
- 'H5Z_FILTER_SHUFFLE'
- 'H5Z_FILTER_FLETCHER32'
- 'H5Z_FILTER_SZIP'
- 'H5Z_FILTER_NBIT'
- 'H5Z_FILTER_SCALEOFFSET'

Examples

Determine if the shuffle filter is available.

```
bool = H5Z.filter_avail('H5Z_FILTER_SHUFFLE');
```

See Also `H5ML.get_constant_value`

Purpose

Information about filter

Syntax

```
filter_config_flags = H5Z.get_filter_info(filter)
```

Description

`filter_config_flags = H5Z.get_filter_info(filter)` retrieves information about the filter specified by its identifier. At present, the information returned is the filter's configuration flags, indicating whether the filter is configured to decode data, to encode data, neither, or both. `filter_config_flags` should be used with the HDF5 constant values `H5Z_FILTER_CONFIG_ENCODE_ENABLED` and `H5Z_FILTER_CONFIG_DECODE_ENABLED` in a bitwise AND operation. If the resulting value is 0, then the encode or decode functionality is not available.

Examples

Determine if encoding is enabled for the deflate filter.

```
flags = H5Z.get_filter_info('H5Z_FILTER_DEFLATE');
functionality = H5ML.get_constant_value('H5Z_FILTER_CONFIG_ENCODE_ENAB
enabled = bitand(flags,functionality) > 0;
```

See Also

`H5Z.filter_avail` | `H5ML.get_constant_value` | `bitand`

hadamard

Purpose

Hadamard matrix

Syntax

```
H = hadamard(n)
```

Description

`H = hadamard(n)` returns the Hadamard matrix of order `n`.

Definitions

Hadamard matrices are matrices of `1`'s and `-1`'s whose columns are orthogonal,

```
H'*H = n*I
```

where `[n n]=size(H)` and `I = eye(n,n)` ,.

They have applications in several different areas, including combinatorics, signal processing, and numerical analysis, [1], [2].

An `n`-by-`n` Hadamard matrix with `n > 2` exists only if `rem(n,4) = 0`. This function handles only the cases where `n`, `n/12`, or `n/20` is a power of 2.

Examples

The command `hadamard(4)` produces the 4-by-4 matrix:

```
1     1     1     1
1    -1     1    -1
1     1    -1    -1
1    -1    -1     1
```

References

[1] Ryser, H. J., *Combinatorial Mathematics*, John Wiley and Sons, 1963.

[2] Pratt, W. K., *Digital Signal Processing*, John Wiley and Sons, 1978.

See Also

`compan | hankel | toeplitz`

Purpose

Abstract class for deriving handle classes

Syntax

`classdef` *`MyHandleClass`* `< handle`

Description

`classdef` *`MyHandleClass`* `< handle` makes *`MyHandleClass`* a subclass of the `handle` class.

The `handle` class is the superclass for all classes that follow handle semantics. A *handle* is a reference to an object. If you copy an object's handle, MATLAB copies only the handle and both the original and copy refer to the same object data. If a function modifies a handle object passed as an input argument, the modification affects the original input object.

In contrast, nonhandle objects (that is, value objects) are not references. Functions must return modified value objects to the caller to cause change to the object outside of the function's workspace.

See "Modifying Objects" for information on passing objects to functions.

Handle object behavior is like that of Handle Graphics objects, where the handle of a graphics object always refers to a particular instance regardless of whether you save the handle when you create the object, store it in another variable, or obtain it with convenience functions like `findobj`, `gca`, and so on.

If you want to create a class that defines events, you must derive that class from the `handle` class.

The `handle` class is an abstract class, so you cannot create an instance of this class directly. You use the `handle` class to derive other classes, which can be concrete classes whose instances are handle objects. See "Handle Classes" for information on using handle classes.

Handle Class Methods

When you derive a class from the `handle` class, your class inherits the following methods.

Method	Purpose
addlistener	Creates a listener for the specified event and assigns a callback function to execute when the event occurs.
notify	Broadcast a notice that a specific event is occurring on a specified handle object or array of handle objects.
delete	Handle object destructor method that is called when the object's lifecycle ends.
findobj	Finds objects matching the specified conditions from the input array of handle objects.
findprop	Returns a `meta.property` objects associated with the specified property name.
isvalid	Returns a logical array in which elements are true if the corresponding elements in the input array are valid handles. This method is `Sealed` so you cannot override it in a handle subclass.
eq ne lt le gt ge relationaloperators	Relational functions return a logical array of the same size as the pair of input handle object arrays. Comparisons use a number associated with each handle. You can assume that the same two handles will compare as equal and the repeated comparison of any two handles will yield the same result in the same MATLAB session. Different handles are always not-equal. The order of handles is purely arbitrary, but consistent.

Handle Class Events

The handle class defines one event:

`ObjectBeingDestroyed`

This event is triggered when the handle object is about to be destroyed. If you define a listener for this event, its callback executes before MATLAB destroys the handle object.

You can add a listener for this event using the `addlistener` method. See "Events and Listeners — Syntax and Techniques" for more information on using events and listeners.

You can define a `delete` method for a handle subclass that MATLAB calls when the object is destroyed. See "Handle Class Destructor".

Handle Subclasses

You can use the following abstract `handle` subclasses to derive more specialized `handle` classes:

- `hgsetget` – use when you want to create a handle class that inherits set and get methods having the same behavior as Handle Graphics `set` and `get` functions.
- `dynamicprops` – use when you want to create a handle class that allows you to add instance data (dynamically defined properties) to objects.
- `matlab.mixin.Copyable` – use to add a copy method to a handle subclass.

Useful Functions

- `properties` — list the class public properties
- `methods` — list the class methods
- `events` — list the events defined by the class

Note that `ishandle` does not test for handle class objects. Use `isa` instead.

hankel

Purpose

Hankel matrix

Syntax

```
H = hankel(c)
H = hankel(c,r)
```

Description

`H = hankel(c)` returns the square Hankel matrix whose first column is `c` and whose elements are zero below the first anti-diagonal.

`H = hankel(c,r)` returns a Hankel matrix whose first column is `c` and whose last row is `r`. If the last element of `c` differs from the first element of `r`, the last element of `c` prevails.

Definitions

A Hankel matrix is a matrix that is symmetric and constant across the anti-diagonals, and has elements `h(i,j) = p(i+j-1)`, where vector `p = [c r(2:end)]` completely determines the Hankel matrix.

Examples

A Hankel matrix with anti-diagonal disagreement is

```
c = 1:3; r = 7:10;
h = hankel(c,r)
h =
    1    2    3    8
    2    3    8    9
    3    8    9   10

p = [1 2 3 8 9 10]
```

See Also

`hadamard` | `toeplitz` | `kron`

Purpose

Summary of MATLAB HDF4 capabilities

Description

The MATLAB software provides a set of low-level functions that enable you to access the HDF4 library developed by the National Center for Supercomputing Applications (NCSA). For information about HDF4, go to the HDF Web page at `http://www.hdfgroup.org`.

Note This section does not apply to MATLAB HDF5 capabilities, which is a completely separate, incompatible format.

The following table lists all the HDF4 application programming interfaces (APIs) supported by MATLAB with the name of the MATLAB function used to access the API. To use these functions, you must be familiar with the HDF library. For more information about using these MATLAB functions, see "Importing Hierarchical Data Format (HDF4) Files" and "Exporting to Hierarchical Data Format (HDF4) Files".

Call `help` *`functionname`* to display help for any of these functions in the Command Window.

Application Programming Interface	Description	MATLAB Function
Annotations	Stores, manages, and retrieves text used to describe an HDF file or any of the data structures contained in the file.	`hdfan`
External Files	Manipulates external file elements.	`hdfhx`

Application Programming Interface	Description	MATLAB Function
General Raster Images	Stores, manages, and retrieves raster images, their dimensions and palettes. It can also manipulate unattached palettes. Note: Use the MATLAB functions `imread` and `imwrite` with HDF raster image formats.	`hdfdf24`, `hdfdfr8`
HDF-EOS	Provides functions to read HDF-EOS grid (GD), point (PT), and swath (SW) data.	`matlab.io.hdfeos.gd`, `hdfpt`, `matlab.io.hdfeos.sw`
HDF Utilities	Provides functions to open and close HDF files and handle errors.	`hdfh`, `hdfhd`, `hdfhe`
MATLAB HDF Utilities	Provides utility functions that help you work with HDF files in the MATLAB environment.	`hdfml`
Scientific Data	Stores, manages, and retrieves multidimensional arrays of character or numeric data, along with their dimensions and attributes.	`matlab.io.hdf4.sd`
V Groups	Creates and retrieves groups of other HDF data objects, such as raster images or V data.	`hdfv`
V Data	Stores, manages, and retrieves multivariate data stored as records in a table.	`hdfvf`, `hdfvh`, `hdfvs`

For copyright information, see the `hdf4copyright.txt` file.

See Also

`hdfinfo` | `hdfread`

hdf5info

Purpose

Information about HDF5 file

Note `hdf5info` will be removed in a future version. Use `h5info` instead.

Syntax

```
fileinfo = hdf5info(filename)
fileinfo = hdf5info(...,'ReadAttributes',BOOL)
[...] = hdf5info(..., 'V71Dimensions', BOOL)
```

Description

`fileinfo = hdf5info(filename)` returns a structure `fileinfo` whose fields contain information about the contents of the HDF5 file `filename`. `filename` is a string that specifies the name of the HDF5 file.

`fileinfo = hdf5info(...,'ReadAttributes',BOOL)` specifies whether `hdf5info` returns the values of the attributes or just information describing the attributes. By default, `hdf5info` reads in attribute values (`BOOL` = `true`).

`[...]  = hdf5info(..., 'V71Dimensions', BOOL)` specifies whether to report the dimensions of data sets and attributes as they were returned in previous versions of `hdf5info` (MATLAB 7.1 [R14SP3] and earlier). If `BOOL` is true, `hdf5info` swaps the first two dimensions of the data set. This behavior was intended to account for the difference in how HDF5 and MATLAB express array dimenions. HDF5 describes data set dimensions in row-major order; MATLAB stores data in column-major order. However, swapping these dimensions may not correctly reflect the intent of the data in the file and may invalidate metadata. When `BOOL` is false (the default), `hdf5info` returns data dimensions that correctly reflect the data ordering as it is written in the file—each dimension in the output variable matches the same dimension in the file.

Note If you use the `'V71Dimensions'` parameter and intend on passing the `fileinfo` structure returned to the `hdf5read` function, you should also specify the `'V71Dimensions'` parameters with `hdf5read`. If you do not, `hdf5read` uses the new behavior when reading the data set and certain metadata returned by `hdf5info` does not match the actual data returned by `hdf5read`.

Examples

```
fileinfo = hdf5info('example.h5');
```

To get more information about the contents of the HDF5 file, look at the `GroupHierarchy` field in the `fileinfo` structure returned by `hdf5info`.

```
toplevel = fileinfo.GroupHierarchy

toplevel =

      Filename: [1x64 char]
          Name: '/'
        Groups: [1x2 struct]
      Datasets: []
     Datatypes: []
         Links: []
    Attributes: [1x2 struct]
```

To probe further into the file hierarchy, keep examining the `Groups` field.

See also

`hdf5read`, `hdf5write`

hdf5read

Purpose

Read HDF5 file

Note `hdf5read` will be removed in a future version. Use `h5read` instead.

Syntax

```
data = hdf5read(filename,datasetname)
attr = hdf5read(filename,attributename)
[data, attr] = hdf5read(...,'ReadAttributes',BOOL)
data = hdf5read(hinfo)
[...] = hdf5read(..., 'V71Dimensions', BOOL)
```

Description

`data = hdf5read(filename,datasetname)` reads all the data in the data set `datasetname` that is stored in the HDF5 file `filename` and returns it in the variable `data`. To determine the names of data sets in an HDF5 file, use the `hdf5info` function.

The return value, `data`, is a multidimensional array. `hdf5read` maps HDF5 data types to native MATLAB data types, whenever possible. If it cannot represent the data using MATLAB data types, `hdf5read` uses one of the HDF5 data type objects. For example, if an HDF5 file contains a data set made up of an enumerated data type, `hdf5read` uses the `hdf5.h5enum` object to represent the data in the MATLAB workspace. The `hdf5.h5enum` object has data members that store the enumerations (names), their corresponding values, and the enumerated data.

Note `hdf5read` performs best when reading numeric datasets. If you need to read string, compound, or variable length datasets, MathWorks strongly recommends that you use the low-level HDF5 interface function, `H5D.read`. To read a subset of a dataset, you must use the low-level interface.

`attr = hdf5read(filename,attributename)` reads all the metadata in the attribute `attributename`, stored in the HDF5 file `filename`, and

returns it in the variable `attr`. To determine the names of attributes in an HDF5 file, use the `hdf5info` function.

`[data, attr] = hdf5read(...,'ReadAttributes',BOOL)` reads all the data, as well as all of the associated attribute information contained within that data set. By default, `BOOL` is false.

`data = hdf5read(hinfo)` reads all of the data in the data set specified in the structure `hinfo` and returns it in the variable `data`. The `hinfo` structure is extracted from the output returned by `hdf5info`, which specifies an HDF5 file and a specific data set.

`[...] = hdf5read(..., 'V71Dimensions', BOOL)` specifies whether to change the majority of data sets read from the file. If `BOOL` is true, `hdf5read` permutes the first two dimensions of the data set, as it did in previous releases (MATLAB 7.1 [R14SP3] and earlier). This behavior was intended to account for the difference in how HDF5 and MATLAB express array dimensions. HDF5 describes data set dimensions in row-major order; MATLAB stores data in column-major order. However, permuting these dimensions may not correctly reflect the intent of the data and may invalidate metadata. When `BOOL` is false (the default), the data dimensions correctly reflect the data ordering as it is written in the file — each dimension in the output variable matches the same dimension in the file.

Examples

Use `hdf5info` to get information about an HDF5 file and then use `hdf5read` to read a data set, using the information structure (`hinfo`) returned by `hdf5info` to specify the data set.

```
hinfo = hdf5info('example.h5');
dset = hdf5read(hinfo.GroupHierarchy.Groups(2).Datasets(1));
```

See Also

`hdf5info` | `hdf5write`

hdf5write

Purpose

Write data to file in HDF5 format

Note `hdf5write` will be removed in a future version. Use `h5write` instead.

Syntax

```
hdf5write(filename,location,dataset)
hdf5write(filename,details,dataset)
hdf5write(filename,details,attribute)
hdf5write(filename, details1, dataset1, details2, dataset2,
   ...)
hdf5write(filename,...,'WriteMode',mode,...)
hdf5write(..., 'V71Dimensions', BOOL)
```

Description

`hdf5write(filename,location,dataset)` writes the data `dataset` to the HDF5 file, `filename`. If `filename` does not exist, `hdf5write` creates it. If `filename` exists, `hdf5write` overwrites the existing file, by default, but you can also append data to an existing file using an optional syntax.

`location` defines where to write the data set in the file. HDF5 files are organized in a hierarchical structure similar to a UNIX directory structure. `location` is a string that resembles a UNIX path.

`hdf5write` maps the data in `dataset` to HDF5 data types according to rules outlined below.

`hdf5write(filename,details,dataset)` writes `dataset` to `filename` using the values in the `details` structure. For a data set, the `details` structure can contain the following fields.

Field Name	Description	Data Type
`Location`	Location of the data set in the file	Character array
`Name`	Name to attach to the data set	Character array

`hdf5write(filename,details,attribute)` writes the metadata `attribute` to `filename` using the values in the `details` structure. For an attribute, the `details` structure can contain following fields.

Field Name	Description	Data Type
`AttachedTo`	Location of the object this attribute modifies	Structure array
`AttachType`	Identifies what kind of object this attribute modifies; possible values are `'group'` and `'dataset'`	Character array
`Name`	Name to attach to the data set	Character array

`hdf5write(filename, details1, dataset1, details2, dataset2,...)` writes multiple data sets and associated attributes to `filename` in one operation. Each data set and attribute must have an associated `details` structure.

`hdf5write(filename,...,'WriteMode',mode,...)` specifies whether `hdf5write` overwrites the existing file (the default) or appends data sets and attributes to the file. Possible values for mode are `'overwrite'` and `'append'`.

`hdf5write(..., 'V71Dimensions', BOOL)` specifies whether to change the majority of data sets written to the file. If `BOOL` is true, `hdf5write` permutes the first two dimensions of the data set, as it did in previous releases (MATLAB 7.1 [R14SP3] and earlier). This behavior was intended to account for the difference in how HDF5 and MATLAB express array dimensions. HDF5 describes data set dimensions in row-major order; MATLAB stores data in column-major order. However, permuting these dimensions may not correctly reflect the intent of the data and may invalidate metadata. When `BOOL` is false (the default), the data written to the file correctly reflects the data ordering of the data sets — each dimension in the file's data sets matches the same dimension in the corresponding MATLAB variable.

Data Type Mappings

The following table lists how `hdf5write` maps the data type from the workspace into an HDF5 file. If the data in the workspace that is being written to the file is a MATLAB data type, `hdf5write` uses the following rules when translating MATLAB data into HDF5 data objects.

MATLAB Data Type	HDF5 Data Set or Attribute
Numeric	Corresponding HDF5 native data type. For example, if the workspace data type is `uint8`, the `hdf5write` function writes the data to the file as 8-bit integers. The size of the HDF5 dataspace is the same size as the MATLAB array.
String	Single, null-terminated string
Cell array of strings	Multiple, null-terminated strings, each the same length. Length is determined by the length of the longest string in the cell array. The size of the HDF5 dataspace is the same size as the cell array.
Cell array of numeric data	Numeric array, the same dimensions as the cell array. The elements of the array must all have the same size and type. The data type is determined by the first element in the cell array.
Structure array	HDF5 compound type. Individual fields in the structure employ the same data translation rules for individual data types. For example, a cell array of strings becomes a multiple, null-terminated strings.
HDF5 objects	If the data being written to the file is composed of HDF5 objects, `hdf5write` uses the same data type when writing to the file. For all HDF5 objects, except `HDF5.h5enum` objects, the dataspace has the same dimensions as the array of HDF5 objects passed to the function. For `HDF5.h5enum` objects, the size and dimensions of the data set in the HDF5 file is the same as the object's Data field.

Examples

Write a 5-by-5 data set of `uint8` values to the root group.

```
hdf5write('myfile.h5', '/dataset1', uint8(magic(5)))
```

Write a 2-by-2 string data set in a subgroup.

```
dataset = {'north', 'south'; 'east', 'west'};
hdf5write('myfile2.h5', '/group1/dataset1.1', dataset);
```

Write a data set and attribute to an existing group.

```
dset = single(rand(10,10));
dset_details.Location = '/group1/dataset1.2';
dset_details.Name = 'Random';

attr = 'Some random data';
attr_details.Name = 'Description';
attr_details.AttachedTo = '/group1/dataset1.2/Random';
attr_details.AttachType = 'dataset';

hdf5write('myfile2.h5', dset_details, dset, ...
          attr_details, attr, 'WriteMode', 'append');
```

Write a data set using objects.

```
dset = hdf5.h5array(magic(5));
hdf5write('myfile3.h5', '/g1/objects', dset);
```

See Also

hdf5read | hdf5info

hdfinfo

Purpose

Information about HDF4 or HDF-EOS file

Syntax

```
S = hdfinfo(filename)
S = hdfinfo(filename,mode)
```

Description

`S = hdfinfo(filename)` returns a structure `S` whose fields contain information about the contents of an HDF4 or HDF-EOS file. `filename` is a string that specifies the name of the HDF4 file.

`S = hdfinfo(filename,mode)` reads the file as an HDF4 file, if `mode` is `'hdf'`, or as an HDF-EOS file, if `mode` is `'eos'`. If `mode` is `'eos'`, only HDF-EOS data objects are queried. To retrieve information on the entire contents of a file containing both HDF4 and HDF-EOS objects, `mode` must be `'hdf'`.

Note `hdfinfo` can be used on Version 4.x HDF files or Version 2.x HDF-EOS files. To get information about an HDF5 file, use `hdf5info`.

The set of fields in the returned structure `S` depends on the individual file. Fields that can be present in the `S` structure are shown in the following table.

Mode	Field Name	Description	Return Type
`HDF`	`Attributes`	Attributes of the data set	Structure array
	`Description`	Annotation description	Cell array
	`Filename`	Name of the file	String
	`Label`	Annotation label	Cell array
	`Raster8`	Description of 8-bit raster images	Structure array

Mode	Field Name	Description	Return Type
	`Raster24`	Description of 24-bit raster images	Structure array
	`SDS`	Description of scientific data sets	Structure array
	`Vdata`	Description of Vdata sets	Structure array
	`Vgroup`	Description of Vgroups	Structure array
`EOS`	`Filename`	Name of the file	String
	`Grid`	Grid data	Structure array
	`Point`	Point data	Structure array
	`Swath`	Swath data	Structure array

Those fields in the table above that contain structure arrays are further described in the tables shown below.

Fields Common to Returned Structure Arrays

Structure arrays returned by `hdfinfo` contain some common fields. These are shown in the table below. Not all structure arrays will contain all of these fields.

Field Name	Description	Data Type
`Attributes`	Data set attributes. Contains fields `Name` and `Value`.	Structure array
`Description`	Annotation description	Cell array
`Filename`	Name of the file	String
`Label`	Annotation label	Cell array

Field Name	Description	Data Type
`Name`	Name of the data set	String
`Rank`	Number of dimensions of the data set	Double
`Ref`	Data set reference number	Double
`Type`	Type of HDF or HDF-EOS object	String

Fields Specific to Certain Structures

Structure arrays returned by `hdfinfo` also contain fields that are unique to each structure. These are shown in the tables below.

Fields of the Attribute Structure

Field Name	Description	Data Type
`Name`	Attribute name	String
`Value`	Attribute value or description	Numeric or string

Fields of the Raster8 and Raster24 Structures

Field Name	Description	Data Type
`HasPalette`	`1` (`true`) if the image has an associated palette, otherwise `0` (`false`) (8-bit only)	Logical
`Height`	Height of the image, in pixels	Number
`Interlace`	Interlace mode of the image (24-bit only)	String
`Name`	Name of the image	String
`Width`	Width of the image, in pixels	Number

Fields of the SDS Structure

Field Name	Description	Data Type
DataType	Data precision	String
Dims	Dimensions of the data set. Contains fields Name, DataType, Size, Scale, and Attributes. Scale is an array of numbers to place along the dimension and demarcate intervals in the data set.	Structure array
Index	Index of the SDS	Number

Fields of the Vdata Structure

Field Name	Description	Data Type
DataAttributes	Attributes of the entire data set. Contains fields Name and Value.	Structure array
Class	Class name of the data set	String
Fields	Fields of the Vdata. Contains fields Name and Attributes.	Structure array
NumRecords	Number of data set records	Double
IsAttribute	1 (true) if Vdata is an attribute, otherwise 0 (false)	Logical

Fields of the Vgroup Structure

Field Name	Description	Data Type
`Class`	Class name of the data set	String
`Raster8`	Description of the 8-bit raster image	Structure array
`Raster24`	Description of the 24-bit raster image	Structure array
`SDS`	Description of the Scientific Data sets	Structure array
`Tag`	Tag of this Vgroup	Number
`Vdata`	Description of the Vdata sets	Structure array
`Vgroup`	Description of the Vgroups	Structure array

Fields of the Grid Structure

Field Name	Description	Data Type
`Columns`	Number of columns in the grid	Number
`DataFields`	Description of the data fields in each `Grid` field of the grid. Contains fields `Name`, `Rank`, `Dims`, `NumberType`, `FillValue`, and `TileDims`.	Structure array
`LowerRight`	Lower right corner location, in meters	Number
`Origin Code`	Origin code for the grid	Number
`PixRegCode`	Pixel registration code	Number

Fields of the Grid Structure (Continued)

Field Name	Description	Data Type
`Projection`	Projection code, zone code, sphere code, and projection parameters of the grid. Contains fields `ProjCode`, `ZoneCode`, `SphereCode`, and `ProjParam`.	Structure
`Rows`	Number of rows in the grid	Number
`UpperLeft`	Upper left corner location, in meters	Number

Fields of the Point Structure

Field Name	Description	Data Type
`Level`	Description of each level of the point. Contains fields `Name`, `NumRecords`, `FieldNames`, `DataType`, and `Index`.	Structure

Fields of the Swath Structure

Field Name	Description	Data Type
`DataFields`	Data fields in the swath. Contains fields `Name`, `Rank`, `Dims`, `NumberType`, and `FillValue`.	Structure array
`GeolocationFields`	Geolocation fields in the swath. Contains fields `Name`,	Structure array

Fields of the Swath Structure (Continued)

Field Name	Description	Data Type
	`Rank`, `Dims`, `NumberType`, and `FillValue`.	
`IdxMapInfo`	Relationship between indexed elements of the geolocation mapping. Contains fields `Map` and `Size`.	Structure
`MapInfo`	Relationship between data and geolocation fields. Contains fields `Map`, `Offset`, and `Increment`.	Structure

Examples

To retrieve information about the file `example.hdf`,

```
fileinfo = hdfinfo('example.hdf')

fileinfo =
    Filename: 'example.hdf'
         SDS: [1x1 struct]
       Vdata: [1x1 struct]
```

And to retrieve information from this about the scientific data set in `example.hdf`,

```
sds_info = fileinfo.SDS

sds_info =
       Filename: 'example.hdf'
           Type: 'Scientific Data Set'
           Name: 'Example SDS'
           Rank: 2
       DataType: 'int16'
     Attributes: []
```

```
          Dims: [2x1 struct]
         Label: {}
   Description: {}
         Index: 0
```

See Also

hdfread | hdf

hdfread

Purpose

Read data from HDF4 or HDF-EOS file

Syntax

```
data = hdfread(filename, datasetname)
data = hdfread(hinfo)
data = hdfread(...,param,value,...)
data = hdfread(filename,EOSname,param,value,...)
[data,map] = hdfread(...)
```

Description

`data = hdfread(filename, datasetname)` returns all the data in the data set specified by `datasetname` from the HDF4 or HDF-EOS file specified by `filename`. To determine the name of a data set in an HDF4 file, use the `hdfinfo` function.

Note `hdfread` can be used on Version 4.x HDF files or Version 2.x HDF-EOS files. To read data from an HDF5 file, use `h5read`.

`data = hdfread(hinfo)` returns all the data in the data set specified by the structure`hinfo`, returned by the `hdfinfo` function. Specify the field in the `hinfo` structure that relates to a particular type of data set, and use indexing to specify which data set, when there are more than one. See "Example 2" on page 1-2460 for more information.

`data = hdfread(...,param,value,...)` returns subsets of the data according to the specified parameter and value pairs. See the tables below to find the valid parameters and values for different types of data sets.

`data = hdfread(filename,EOSname,param,value,...)` subsets the data field from the HDF-EOS point, grid, or swath specified by `EOSname`.

`[data,map] = hdfread(...)` returns the image `data` and the colormap `map` for an 8-bit raster image.

Subsetting Parameters

The following tables show the subsetting parameters that can be used with the `hdfread` function for certain types of HDF4 data. These data types are

- HDF Scientific Data (SD)
- HDF Vdata (V)
- HDF-EOS Grid Data
- HDF-EOS Point Data
- HDF-EOS Swath Data

Note the following:

- If a parameter requires multiple values, use a cell array to store the values. For example, the `'Index'` parameter requires three values: `start`, `stride`, and `edge`. Enclose these values in curly braces as a cell array.

  ```
  hdfread(..., 'Index', {start,stride,edge})
  ```

- All values that are indices are 1-based.

Subsetting Parameters for HDF Scientific Data (SD) Data Sets

When you are working with HDF SD files, `hdfread` supports the parameters listed in this table.

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>'Index'</td><td>Three-element cell array, {start,stride,edge}, specifying the location, range, and values to be read from the data set

• start — A 1-based array specifying the position in the file to begin reading
Default: 1, start at the first element of each dimension. The values specified must not exceed the size of any dimension of the data set.
• stride — A 1-based array specifying the interval between the values to read
Default: 1, read every element of the data set.
• edge — A 1-based array specifying the length of each dimension to read
Default: An array containing the lengths of the corresponding dimensions</td></tr>
</table>

For example, this code reads the data set Example SDS from the HDF file example.hdf. The 'Index' parameter specifies that hdfread start reading data at the beginning of each dimension, read until the end of each dimension, but only read every other data value in the first dimension.

```
data = hdfread('example.hdf','Example SDS','Index',{[],[2 1],[]})
```

Subsetting Parameters for HDF Vdata Sets

When you are working with HDF Vdata files, hdfread supports these parameters.

Parameter	Description
'Fields'	Text string specifying the name of the field to be read. When specifying multiple field names, use a cell array.
'FirstRecord'	1-based number specifying the record from which to begin reading
'NumRecords'	Number specifying the total number of records to read

For example, this code reads the Vdata set `Example Vdata` from the HDF file `example.hdf`.

```
data = hdfread('example.hdf','Example Vdata','FirstRecord', 2,'NumRecords', 5)
```

Subsetting Parameters for HDF-EOS Grid Data

When you are working with HDF-EOS grid data, `hdfread` supports three types of parameters:

- Required parameters
- Optional parameters
- Mutually exclusive parameters — You can only specify one of these parameters in a call to `hdfread`, and you cannot use these parameters in combination with any optional parameter.

Parameter	Description
Required Parameter	
`'Fields'`	String specifying the field to be read. You can specify only one field name for a Grid data set.
Mutually Exclusive Optional Parameters	
`'Index'`	Three-element cell array, `{start,stride,edge}`, specifying the location, range, and values to be read from the data set `start` — An array specifying the position in the file to begin reading Default: `1`, start at the first element of each dimension. The values must not exceed the size of any dimension of the data set. `stride` — An array specifying the interval between the values to read Default: `1`, read every element of the data set. `edge` — An array specifying the length of each dimension to read Default: An array containing the lengths of the corresponding dimensions

Parameter	Description
`'Interpolate'`	Two-element cell array, `{longitude,latitude}`, specifying the longitude and latitude points that define a region for bilinear interpolation. Each element is an N-length vector specifying longitude and latitude coordinates.
`'Pixels'`	Two-element cell array, `{longitude,latitude}`, specifying the longitude and latitude coordinates that define a region. Each element is an N-length vector specifying longitude and latitude coordinates. This region is converted into pixel rows and columns with the origin in the upper left corner of the grid. Note: This is the pixel equivalent of reading a `'Box'` region.
`'Tile'`	Vector specifying the coordinates of the tile to read, for HDF-EOS Grid files that support tiles
Optional Parameters	
`'Box'`	Two-element cell array, `{longitude,latitude}`, specifying the longitude and latitude coordinates that define a region. `longitude` and `latitude` are each two-element vectors specifying longitude and latitude coordinates.
`'Time'`	Two-element cell array, `[start stop]`, where `start` and `stop` are numbers that specify the start and end-point for a period of time
`'Vertical'`	Two-element cell array, `{dimension, range}` `dimension` — String specifying the name of the data set field to be read from. You can specify only one field name for a Grid data set. `range` — Two-element array specifying the minimum and maximum range for the subset. If `dimension` is a dimension name, then `range` specifies the range of elements to extract. If `dimension` is a field name, then `range` specifies the range of values to extract. `'Vertical'` subsetting can be used alone or in conjunction with `'Box'` or `'Time'`. To subset a region along multiple dimensions, vertical subsetting can be used up to eight times in one call to `hdfread`.

For example,

```
data = hdfread('grid.hdf','PolarGrid','Fields','ice_temp','Index', {[5 10],[],[15 20]})
```

Subsetting Parameters for HDF-EOS Point Data

When you are working with HDF-EOS Point data, `hdfread` has two required parameters and three optional parameters.

Parameter	Description
Required Parameters	
`'Fields'`	String naming the data set field to be read. For multiple field names, use a comma-separated list.
`'Level'`	1-based number specifying which level to read from in an HDF-EOS Point data set
Mutually Exclusive Optional Parameters	
`'Box'`	Two-element cell array, `{longitude,latitude}`, specifying the longitude and latitude coordinates that define a region. `longitude` and `latitude` are each two-element vectors specifying longitude and latitude coordinates.
`'RecordNumbers'`	Vector specifying the record numbers to read
`'Time'`	Two-element cell array, `[start stop]`, where `start` and `stop` are numbers that specify the start and endpoint for a period of time

For example,

```
hdfread(...,'Fields',{field1, field2},...
          'Level',level,'RecordNumbers',[1:50, 200:250])
```

Subsetting Parameters for HDF-EOS Swath Data

When you are working with HDF-EOS Swath data, `hdfread` supports three types of parameters:

- Required parameters
- Optional parameters

- Mutually exclusive

You can only use one of the mutually exclusive parameters in a call to `hdfread`, and you cannot use these parameters in combination with any optional parameter.

Parameter	Description
Required Parameter	
`'Fields'`	String naming the data set field to be read. You can specify only one field name for a Swath data set.
Mutually Exclusive Optional Parameters	
`'Index'`	Three-element cell array, `{start,stride,edge}`, specifying the location, range, and values to be read from the data set • `start` — An array specifying the position in the file to begin reading Default: `1`, start at the first element of each dimension. The values must not exceed the size of any dimension of the data set. • `stride` — An array specifying the interval between the values to read Default: `1`, read every element of the data set. • `edge` — An array specifying the length of each dimension to read Default: An array containing the lengths of the corresponding dimensions
`'Time'`	Three-element cell array, `{start, stop, mode}`, where `start` and `stop` specify the beginning and the endpoint for a period of time, and `mode` is a string defining the criterion for the inclusion of a cross track in a region. The cross track is within a region if any of these conditions is met: • Its midpoint is within the box (`mode='midpoint'`). • Either endpoint is within the box (`mode='endpoint'`).

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td colspan="2">Optional Parameters</td></tr>
<tr><td><code>'Box'</code></td><td>Three-element cell array, <code>{longitude, latitude, mode}</code> specifying the longitude and latitude coordinates that define a region. <code>longitude</code> and <code>latitude</code> are two-element vectors that specify longitude and latitude coordinates. <code>mode</code> is a string defining the criterion for the inclusion of a cross track in a region. The cross track is within a region if any of these conditions is met:
Its midpoint is within the box (<code>mode='midpoint'</code>).
Either endpoint is within the box (<code>mode='endpoint'</code>).
Any point is within the box (<code>mode='anypoint'</code>).</td></tr>
<tr><td><code>'Vertical'</code></td><td>Two-element cell array, <code>{dimension, range}</code>
<code>dimension</code> is a string specifying either a dimension name or field name to subset the data by.
<code>range</code> is a two-element vector specifying the minimum and maximum range for the subset. If <code>dimension</code> is a dimension name, then <code>range</code> specifies the range of elements to extract. If <code>dimension</code> is a field name, then <code>range</code> specifies the range of values to extract.

<code>'Vertical'</code> subsetting can be used alone or in conjunction with <code>'Box'</code> or <code>'Time'</code>. To subset a region along multiple dimensions, vertical subsetting can be used up to eight times in one call to <code>hdfread</code>.</td></tr>
</table>

For example,

```
hdfread('swath.hdf', 'Example Swath', 'Fields', 'Temperature', ...
        'Time', {5000, 6000, 'midpoint'})
```

Examples

Example 1

Specify the name of the HDF file and the name of the data set. This example reads a data set named 'Example SDS' from a sample HDF file.

```
data = hdfread('sd.hdf','temperature')
```

Example 2

Use data returned by hdfinfo to specify the data set to read.

1 Call hdfinfo to retrieve information about the contents of the HDF file.

```
fileinfo = hdfinfo('sd.hdf')
fileinfo =

    Filename: 'C:Program Files\matlab\toolbox\matlab\imagesci\sd.hdf'
  Attributes: [1x1 struct]
         SDS: [1x2 struct]
       Vdata: [1x1 struct]
```

2 Extract the structure containing information about the particular data set you want to import from the data returned by hdfinfo. The example uses the structure in the SDS field to retrieve a scientific data set.

```
sds_info = fileinfo.SDS(2)

sds_info =

       Filename: [1x73 char]
           Type: 'Scientific Data Set'
           Name: 'temperature'
           Rank: 2
       DataType: 'double'
     Attributes: [1x11 struct]
           Dims: [2x1 struct]
          Label: {}
```

```
    Description: {}
          Index: 1
```

3 Pass this structure to `hdfread` to import the data in the data set.

```
data = hdfread(sds_info)
```

Example 3

You can use the information returned by `hdfinfo` to check the size of the data set.

```
sds_info.Dims.Size
ans =
    16
ans =
     5
```

Using the `'index'` parameter with `hdfread`, you can read a subset of the data in the data set. This example specifies a starting index of `[3 3]`, an interval of 1 between values (`[]` meaning the default value of 1), and a length of 10 rows and 2 columns.

```
data = hdfread(sds_info, 'Index', {[3 3],[],[10 2]});

data(:,1)
ans =
     7
     8
     9
    10
    11
    12
    13
    14
    15
    16

data(:,2)
```

```
ans =
     8
     9
    10
    11
    12
    13
    14
    15
    16
    17
```

Example 4

This example uses the `Vdata` field from the information returned by `hdfinfo` to read two fields of the data, `Idx` and `Temp`.

```
s = hdfinfo('example.hdf');
data1 = hdfread(s.Vdata(1), 'Fields', {'Idx', 'Temp', 'Dewpt'})

data1 =

    [1x10 int16]
    [1x10 int16]
    [1x10 int16]
```

See Also

`hdfinfo` | `hdf`

Purpose

Browse and import data from HDF4 or HDF-EOS files

Note `hdftool` will be removed in a future release. To read data from an HDF file, use `hdfread`, `matlab.io.hdf4.sd`, `matlab.io.hdfeos.gd`, or `matlab.io.hdfeos.sw` instead.

Syntax

```
hdftool
hdftool(filename)
h = hdftool(...)
```

Description

`hdftool` starts the HDF Import Tool, a graphical user interface used to browse the contents of HDF4 and HDF-EOS files and import data and subsets of data from these files. To open an HDF4 or HDF-EOS file, select **Open** from the **Home** tab. You can open multiple files in the HDF Import Tool by selecting **Open** from the **Home** tab.

`hdftool(filename)` opens the HDF4 or HDF-EOS file specified by `filename` in the HDF Import Tool.

`h = hdftool(...)` returns a handle `h` to the HDF Import Tool. To close the tool from the command line, use `close(h)`.

Examples

```
hdftool('example.hdf');
```

See Also

`hdf` | `hdfinfo` | `hdfread` | `uiimport`

help

Purpose

Help for functions in Command Window

Syntax

```
help
help name
```

Description

`help` lists all primary help topics in the Command Window. Each main help topic corresponds to a folder name on the MATLAB search path.

`help name` displays the help text for the functionality specified by `name`, such as a function, method, class, or toolbox.

Tips

- Some help text displays the names of functions in uppercase characters to make them stand out from the rest of the text. When typing these function names, use lowercase. For function names that appear in mixed case (such as `javaObject`), type the names as shown.
- To prevent long descriptions from scrolling off the screen before you have time to read them, enter `more on`, and then enter the `help` statement.
- Some classes require that you specify the package name to display the help text. To identify the package name, create an instance of the class, and then call `class(obj)`.

Input Arguments

name

String that specifies an operator symbol (such as +) or the name of a function, class, method, package, toolbox folder, or other functionality.

Some classes require that you specify the package name. Events, properties, and some methods require that you specify the class name. Separate the components of the name with periods, using one of the following forms:

```
help className.name
help packageName.className
help packageName.className.name
```

If `name` is overloaded, that is, appears in multiple folders on the search path, `help` displays the help text for the first instance of `name` found on the search path, and displays a hyperlinked list of the overloaded functions and their folders.

When `name` specifies the name or partial path of a toolbox folder:

- If the folder contains a nonempty `Contents.m` file, the `help` function displays the file. `Contents.m` contains a list of MATLAB program files in the folder and their descriptions. If `Contents.m` exists, but is empty, MATLAB responds with `No help found for` *`name`*.
- If the folder does not contain a `Contents.m` file, the `help` function lists the first line of help text for each program file in the folder.
- If `name` is the name of both a function and a toolbox, `help` displays the associated text for both the toolbox and the function.

Examples

Functions and Overloaded Methods

Display help for the MATLAB `close` function.

```
help close
```

Because `close` refers to the name of a function and to the name of several methods, the help text includes hyperlinks to the overloaded methods.

Request help for the Database Toolbox™ `close` method.

```
help database.close
```

Package, Class, and Method Help

Display help for the `containers` package, `Map` class, and the `isKey` method.

```
help containers
help containers.Map
help containers.Map.isKey
```

Not all packages, classes, and associated methods or events require complete specification. For example, display the help for the `throwAsCaller` method of the `MException` class.

```
help throwAsCaller
```

Functions in Folder

List all of the functions in the folder *matlabroot*`/toolbox/matlab/general` by specifying a partial path.

```
help general
```

Alternatives

View more extensive help using the `doc` command or the Function Browser. To open the Function Browser, click its icon, *fx*.

See Also

`class` | `dbtype` | `doc` | `lookfor` | `more` | `path` | `what` | `which` | `whos`

How To

- "Ways to Get Function Help"
- "Add Help for Your Program"

helpbrowser

Purpose

Open Help browser to access online documentation

Note `helpbrowser` will be removed in a future release. Use `doc` instead.

Syntax

```
helpbrowser
```

Description

`helpbrowser` displays the Help browser, open to its default startup page.

See Also

`doc` | `help`

How To

- "Ways to Get Function Help"

helpdesk

Purpose

Open Help browser

Note `helpdesk` will be removed in a future release. Use `doc` instead.

Syntax

```
helpdesk
```

Description

`helpdesk` opens the Help browser to the default startup page. In previous releases, `helpdesk` displayed the Help Desk, which was the precursor to the Help browser.

See Also

`doc`

Purpose

Create and open help dialog box

Syntax

```
helpdlg
helpdlg('helpstring')
helpdlg('helpstring','dlgname')
h = helpdlg(...)
```

Description

`helpdlg` creates a nonmodal help dialog box or brings the named help dialog box to the front.

Note A nonmodal dialog box enables the user to interact with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

`helpdlg` displays a dialog box named `'Help Dialog'` containing the string `'This is the default help string.'`

`helpdlg('helpstring')` displays a dialog box named `'Help Dialog'` containing the string specified by `'helpstring'`.

`helpdlg('helpstring','dlgname')` displays a dialog box named `'dlgname'` containing the string `'helpstring'`.

`h = helpdlg(...)` returns the handle of the dialog box.

Tips

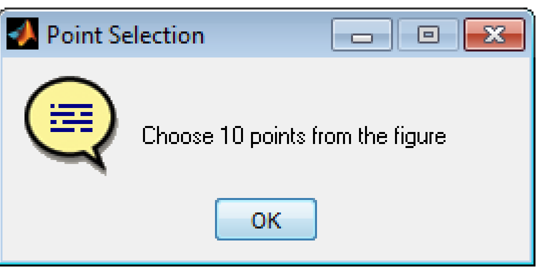
MATLAB wraps the text in `'helpstring'` to fit the width of the dialog box. The dialog box remains on your screen until you press the **OK** button or the **Enter** key. After either of these actions, the help dialog box disappears.

Examples

The statement

```
helpdlg('Choose 10 points from the figure','Point Selection');
```

displays this dialog box:



See Also

`dialog` | `errordlg` | `inputdlg` | `listdlg` | `msgbox` | `questdlg` | `warndlg` | `figure` | `uiwait` | `uiresume`

Purpose

Provide access to help comments for all functions

Note `helpwin` will be removed in a future release. Use `doc` instead.

Syntax

```
helpwin
helpwin topic
```

Description

`helpwin` lists topics for groups of functions in the MATLAB Web browser. It shows brief descriptions of the topics and provides links to display help comments for the functions. You cannot follow links in the `helpwin` list of functions if the MATLAB software is busy (for example, running a program).

`helpwin topic` displays help information for the topic. If `topic` is a folder, it displays all functions in the folder. If `topic` is a function, `helpwin` displays help for that function. From the page, you can access a list of folders (**Default Topics** link) as well as the reference page help for the function (**Go to online doc** link). You cannot follow links in the `helpwin` list of functions if MATLAB is busy (for example, running a program).

Examples

Typing

```
helpwin datafun
```

displays the functions in the `datafun` folder and a brief description of each.

Typing

```
helpwin fft
```

displays the help for the `fft` function.

See Also

`doc` | `help`

hess

Purpose

Hessenberg form of matrix

Syntax

```
H = hess(A)
[P,H] = hess(A)
[AA,BB,Q,Z] = hess(A,B)
```

Description

`H = hess(A)` finds `H`, the Hessenberg form of matrix `A`.

`[P,H] = hess(A)` produces a Hessenberg matrix `H` and a unitary matrix `P` so that `A = P*H*P'` and `P'*P = eye(size(A))`.

`[AA,BB,Q,Z] = hess(A,B)` for square matrices `A` and `B`, produces an upper Hessenberg matrix `AA`, an upper triangular matrix `BB`, and unitary matrices `Q` and `Z` such that `Q*A*Z = AA` and `Q*B*Z = BB`.

Definitions

A Hessenberg matrix is zero below the first subdiagonal. If the matrix is symmetric or Hermitian, the form is tridiagonal. This matrix has the same eigenvalues as the original, but less computation is needed to reveal them.

Examples

`H` is a 3-by-3 eigenvalue test matrix:

```
H =
   -149     -50    -154
    537     180     546
    -27      -9     -25
```

Its Hessenberg form introduces a single zero in the (3,1) position:

```
hess(H) =
   -149.0000    42.2037   -156.3165
   -537.6783   152.5511   -554.9272
           0     0.0728      2.4489
```

See Also

`eig` | `qz` | `schur`

Purpose

Superclass for heterogeneous array formation

Description

`matlab.mixin.Heterogeneous` is an abstract class that provides support for the formation of heterogeneous arrays. A heterogeneous array is an array of objects that differ in their specific class, but are all derived from or are instances of a root class. The root class derives directly from `matlab.mixin.Heterogeneous`.

Heterogeneous Hierarchy

Use `matlab.mixin.Heterogeneous` to define hierarchies of classes whose instances you can combine into heterogeneous arrays.

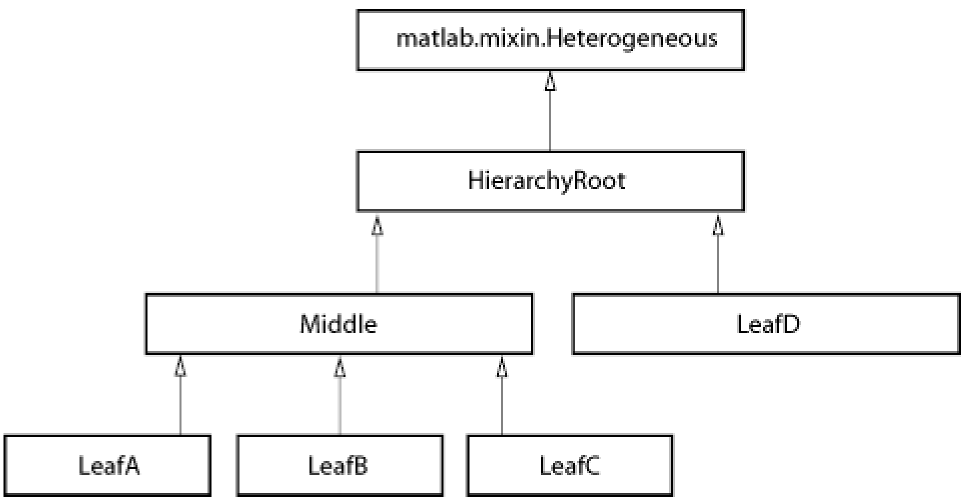
The following class definition enables the formation of heterogeneous arrays that combine instances of any classes derived from `HierarchyRoot`.

```
classdef HierarchyRoot < matlab.mixin.Heterogeneous
% HierarchyRoot is a direct subclass of matlab.mixin.Heterogeneous
% HierarchyRoot is the root of this heterogeneous hierarchy
```

Deriving the `HierarchyRoot` class directly from `matlab.mixin.Heterogeneous` enables the `HierarchyRoot` class to become the root of a hierarchy of classes. You can combine instances of the members of this hierarchy into a heterogeneous array. Only instances of classes derived from the same root class can combine to form a valid heterogeneous array.

Class of a Heterogeneous Array

The class of a heterogeneous array is always the class of the most specific superclass common to all objects in the array. For example, suppose you define the following class hierarchy:



Forming an array (`harray`) of an instance of `LeafA` with an instance of `LeafB` creates an array of class `Middle`:

```
harray = [LeafA,LeafB];
class(harray)
ans =
Middle
```

Forming an array (`harray`) of an instance of `LeafC` with an instance of `LeafD` creates an array of class `HierarchyRoot`:

```
harray = [LeafC,LeafD];
class(harray)
ans =
HierarchyRoot
```

Forming an array (`harray`) of an instance of `LeafA` with an other instance of `LeafA` creates a homogeneous array of class `LeafA`:

```
harray = [LeafA,LeafA];
class(harray)
ans =
```

```
LeafA
```

Note You cannot form heterogeneous arrays that include instances of classes that are not derived from the same hierarchy root (that is, the `HierarchyRoot` class in the hierarchy shown previously).

Forming a Heterogeneous Array

Heterogeneous arrays are the result of operations that produces arrays containing instances of two or more classes from the heterogeneous hierarchy. Usually, concatenation or indexed assignment form these arrays. For example, these statements form `harray` using indexed assignment:

```
harray(1) = LeafA;
harray(2) = LeafC;
class(harray)
ans =
Middle
```

Growing the Array Can Change Its Class

Assigning new objects into an array containing objects derived from `matlab.mixin.Heterogeneous` can change the class of the array. For example, given a homogeneous array containing objects only of the `LeafA` class:

```
harray = [LeafA,LeafA,LeafA];
class(harray)
ans =
LeafA
```

Adding an object of another class derived from the same root to `harray` converts the array's class to the most specific superclass:

```
harray(4) = LeafB;
class(harray)
ans =
```

```
Middle
```

Method Dispatching

When MATLAB invokes a method for which the dominant argument is a heterogeneous array, the method:

- Must be defined for the class of the heterogeneous array, either directly by the class of the array or inherited from a superclass.
- Must be `Sealed = true` (cannot be overridden by a subclass).

The class of the heterogeneous array determines which class method executes for any given method invocation, as is the case with a homogeneous array. MATLAB does not consider the class of individual elements in the array when dispatching to methods.

Sealing Inherited Methods

The requirement that methods called on a heterogeneous array be `Sealed = true` ensures correct and predictable behavior with all array elements.

You must override methods that are inherited from outside the heterogeneous hierarchy if these methods are not `Sealed = true` and you want to call these methods on heterogeneous arrays.

For example, suppose you define a heterogeneous array by subclassing `hgsetget`, in addition to `matlab.mixin.Heterogeneous`. Override the `set` method to call the `hgsetget` superclass method as required by your class design:

```
methods(Sealed)
   function varargout = set(obj,varargin)
      if nargout == 0
         set@hgsetget(obj, varargin{:}); % Call superclass set method
      else
         varargout{:} = set@hgsetget(obj,varargin{:});
      end
   end
end
```

Method implementations can take advantage of the fact that, given a heterogeneous array `harray`, and a scalar index `n`, the expression

```
harray(n)
```

is not a heterogeneous array. Therefore, when invoking a method on a single element of a heterogeneous array, special requirements for heterogeneous arrays do not apply.

Defining the Default Object

When working with object arrays (both heterogeneous and homogeneous), MATLAB creates default objects to fill in missing array elements by calling the class constructor with no arguments. Filling in missing array elements becomes necessary in cases such as:

- Indexed assignment creates an array with gaps. For example, if `harray` is not previously defined:

  ```
  harray(5) = LeafA;
  ```

- Loading a heterogeneous array from a MAT-file when MATLAB cannot find the class definition of a specific object.

The `matlab.mixin.Heterogeneous` class provides a default implementation of a method called `getDefaultScalarElement`. This method returns an instance of the root class of the heterogeneous hierarchy, unless the root class is abstract.

If the root class is abstract or is not an appropriate default object for the classes in the heterogeneous hierarchy, you can override the `getDefaultScalarElement` method to return an instance of a class derived from the root class.

Defining the getDefaultScalarElement Method

Specify the class of the default object by overriding the `matlab.mixin.Heterogeneous` method called `getDefaultScalarElement` in the root class of the heterogeneous

hierarchy. You can override `getDefaultScalarElement` only in the root class (direct subclasses of `matlab.mixin.Heterogeneous`).

`getDefaultScalarElement` has the following signature:

```
methods (Static, Sealed, Access = protected)
   function default_object = getDefaultScalarElement
   ...
   end
end
```

The `getDefaultScalarElement` method must satisfy these criteria:

- Static — MATLAB calls this method without an object.
- Protected — MATLAB calls this method, object users do not.
- Sealed (not required) — Seal this method to ensure users of the heterogeneous hierarchy do not change the intended behavior of the class.
- It must return a scalar object
- It must pass the `isa` test for the root class, that is:

  ```
  (isa(getDefaultScalarElement,'HierarchyRoot')
  ```

  where *`HierarchyRoot`* is the name of the heterogeneous hierarchy root class. This means the default object can be an instance of any class derived from the root class.

Cannot Redefine Indexing or Concatenation

The use of heterogeneous arrays requires consistent indexing and concatenation behaviors. Therefore, subclasses of `matlab.mixin.Heterogeneous` cannot change their default indexed reference, indexed assignment, or concatenation behavior.

You cannot override the following methods in your subclasses:

- `cat`

- `horzcat`
- `vertcat`
- `subref`
- `subsasign`

In cases involving multiple inheritance in which your subclass inherits from superclasses in addition to `matlab.mixin.Heterogeneous`, the superclasses cannot define any of these methods.

Default Concatenation Behavior

Statements of the form:

```
a = [obj1,obj2,...];
```

create an array, `a`, containing the objects listed in brackets.

Concatenating `Heterogeneous` objects of the same specific class retains the class of the objects and does not form a heterogeneous array.

Concatenating `Heterogeneous` objects derived from the same root superclass, but that are of different specific classes, yields a heterogeneous array. MATLAB does not attempt to convert the class of any array members if all are part of the same root hierarchy.

Indexed Assignment Behavior

Statements of the form:

```
a(m:n) = [objm,...objn];
```

assign the right-hand side objects to the array elements (`m:n`), specified on the left side of the assignment.

Indexed assignment to a heterogeneous array can:

- Increase or decrease the size of the array
- Overwrite existing array elements
- Change property values of objects within the array

- Change the class of the array
- Change whether the array is heterogeneous

Indexed Reference Behavior

Statements of the form:

```
a = harray(m:n);
```

assign the elements of `harray` referenced by indices `m:n`, to array `a`.

Indexed reference on a heterogeneous array returns a sub-range of the original array. Depending on the specific elements within that sub-range (`m:n`), the result might have a different class than the original array, and might not be heterogeneous.

Converting Nonmember Objects

If you attempt to form a heterogeneous array with objects that are not derived from the same root class, MATLAB calls the `convertObject` method, if it exists, to convert objects to the dominant class. Implementing a `convertObject` method enables the formation of heterogeneous arrays containing objects that are not part of the heterogeneous hierarchy.

When Is Conversion Necessary

Suppose there are two classes `A` and `B`, where `B` is not derived from `matlab.mixin.Heterogeneous`, or where `A` and `B` are derived from different root classes that are derived from `matlab.mixin.Heterogeneous`.

MATLAB attempts to call the `convertObject` method implemented by the class of `A` in the following cases:

- The indexed assignment:

  ```
  A(k) = B
  ```

- Horizontal and vertical concatenations:

  ```
  [A,B] and [A;B]
  ```

Implement a `convertObject` method if you want to support conversion of objects whose class is not defined in your heterogeneous hierarchy. You do not need to implement this method if your class design does not require this conversion.

Implementing convertObject

The `convertObject` method must have the following signature:

```
Method (Static, Protected, Sealed)
   function converted_object = convertObject('DominantClassName',object
   ...
   end
end
```

For indexed assignment `A(k) = B` and concatenation `[A,B]`:

- *`DominantClassName`* — Name of the class of the array `A`
- `objectToConvert` — Object to be converted, `B` in this case
- `converted_object` — Legal member of the heterogeneous hierarchy to which `A` belongs

You must implement `convertObject` to return a valid object of class `A` or MATLAB issues an error.

Handle Compatibility

The `matlab.mixin.Heterogeneous` class is handle compatible. It can be combined with either handle or value classes when defining a subclass using multiple superclasses. See "Supporting Both Handle and Value Subclasses" for information on handle compatibility.

The `matlab.mixin.Heterogeneous` class is a value class. To learn how value classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation

Methods

cat	Concatenation for heterogeneous arrays
getDefaultScalarElement	Return default object for heterogeneous array operations
horzcat	Horizontal concatenation for heterogeneous arrays
vertcat	Vertical concatenation for heterogeneous arrays

See Also

`handle`

How To

- Class Attributes
- Property Attributes
- "Creating Subclasses — Syntax and Techniques"

Purpose

Concatenation for heterogeneous arrays

Syntax

```
C = cat(dim,A,B)
```

Description

`C = cat(dim,A,B)` concatenates objects `A` and `B` along the dimension `dim`. The class of object arrays `A` and `B` must be derived from the same root class of a `matlab.mixin.Heterogeneous` hierarchy.

- If `A` and `B` are of the same class, the class of the resulting array is unchanged.
- If `A` and `B` are of different subclasses of a common superclass that is derived from `matlab.mixin.Heterogeneous`, then the result is a heterogeneous array and the array's class is that of the most specific superclass shared by `A` and `B`.

The `cat` method is sealed in the class `matlab.mixin.Heterogeneous` and, therefore, you cannot override it in subclasses.

Input Arguments

dim

Scalar dimension along which to concatenate arrays

A

Object array derived from the same root subclass of `matlab.mixin.Heterogeneous` as `B`

B

Object array derived from the same root subclass of `matlab.mixin.Heterogeneous` as `A`

Output Arguments

C

Array resulting from the specified concatenation. The class of this array is that of the most specific superclass shared by `A` and `B`.

Attributes

`Sealed`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.Heterogeneous` | `cat`

matlab.mixin.Heterogeneous.getDefaultScalarElement

Purpose

Return default object for heterogeneous array operations

Syntax

```
defaultObject = getDefaultScalarElement
```

Description

`defaultObject = getDefaultScalarElement` returns the default object for a heterogeneous hierarchy. Override this method if the "Root Class" on page 1-2486 is abstract or is not an appropriate default object for the classes in the heterogeneous hierarchy. `getDefaultScalarElement` must return an instance of another member of the heterogeneous hierarchy.

The `matlab.mixin.Heterogeneous` class provided a default implementation of this method that returns an instance of the "Root Class" on page 1-2486.

MATLAB calls the `getDefaultScalarElement` method when requiring a default object. See `matlab.mixin.Heterogeneous` for more information on heterogeneous arrays and default objects.

Tips

- Override `getDefaultScalarElement` only if the "Root Class" on page 1-2486 is not suitable as a default object.
- Override `getDefaultScalarElement` only in the "Root Class" on page 1-2486 of the heterogeneous hierarchy.
- `getDefaultScalarElement` must return a scalar object.
- `getDefaultScalarElement` must be a static method with protected access. While not required by MATLAB, you can seal this method to prevent overriding by other classes.
- MATLAB returns an error if the value returned by `getDefaultScalarElement` is not scalar or is not an instance of a class that is a valid member of the hierarchy.

Output Arguments

defaultObject

The default object for heterogeneous array operations.

Definitions

Root Class

Root class – The direct subclass of `matlab.mixin.Heterogeneous` that forms the root of a heterogeneous hierarchy. Classes of objects that you can combine into heterogeneous arrays must derive from this root class.

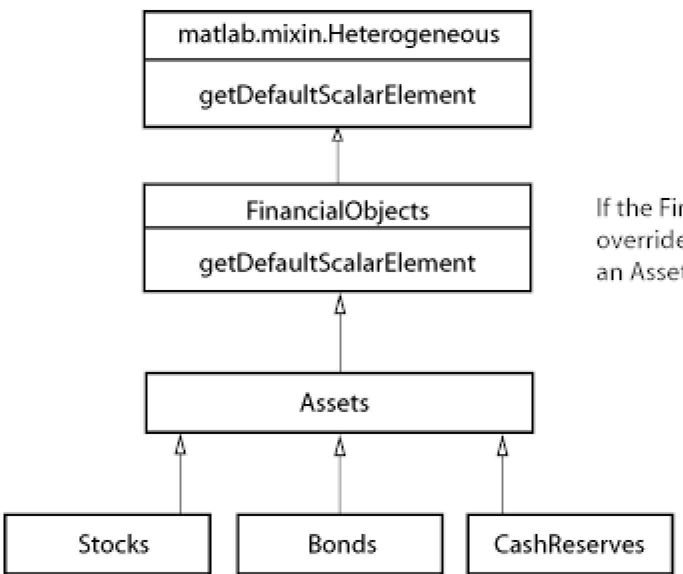
Attributes

`Static`	`true`
`Access`	`Protected`
`Sealed`	`true` not required

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

This example describes a heterogeneous hierarchy with a root class (`FinancialObjects`) that is an abstract class and cannot, therefore, be used for the default object. The `FinancialObjects` class definition includes an override of the `getDefaultScalarElement` method which returns an instance of the `Assets` class as the default object.



The root class can override the `getDefaultScalarElement` method that is defined in `matlab.mixin.Heterogeneous` class and return an `Assets` object as the default object.

```
classdef FinancialObjects < matlab.mixin.Heterogeneous
   methods (Abstract)
      val = determineCurrentValue(obj)
   end
   methods (Static, Sealed, Access = protected)
      function default_object = getDefaultScalarElement
         default_object = Assets;
      end
   end
end
```

See Also

`matlab.mixin.Heterogeneous`

matlab.mixin.Heterogeneous.horzcat

Purpose

Horizontal concatenation for heterogeneous arrays

Syntax

```
C = horzcat(A1,A2,...)
```

Description

`C = horzcat(A1,A2,...)` concatenates the `matlab.mixin.Heterogeneous` objects `A1`, `A2`, and so on, to form the array `C`. All input arrays must have the same number of rows.

The class of object arrays `A1,A2,...` must be derived from the same root class of a `matlab.mixin.Heterogeneous` hierarchy.

MATLAB calls:

```
C = horzcat(A1,A2,...)
```

for the expressions:

```
C = [A1,A2,...]
C = [A1 A2 ...]
```

when `A1` is an array of `matlab.mixin.Heterogeneous` objects.

If all input arguments are of the same specific class, the class of the resulting array is unchanged. If all input arguments are of different subclasses of a common superclass that is derived from `matlab.mixin.Heterogeneous`, then the result is a heterogeneous array. The array's class is that of the most specific superclass shared by all input arguments.

If all input arguments are not members of the same heterogeneous hierarchy, MATLAB calls the `convertObjects` method, if defined by the dominant root class (the first argument or the left-most element in the concatenation if no other class is dominant).

The `horzcat` method is sealed in the class `matlab.mixin.Heterogeneous` and, therefore, you cannot override it in subclasses.

Input Arguments

A1

Object array of class `matlab.mixin.Heterogeneous`

A2

Object array of class `matlab.mixin.Heterogeneous`

Output Arguments

C

Array resulting from the specified concatenation. The class of this array is that of the most specific superclass shared by the input arguments.

Attributes

`Sealed`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.Heterogeneous` | `horzcat`

matlab.mixin.Heterogeneous.vertcat

Purpose Vertical concatenation for heterogeneous arrays

Syntax

```
C = vertcat(A1,A2,...)
```

Description

`C = vertcat(A1,A2,...)` concatenates the `matlab.mixin.Heterogeneous` objects `A1`, `A2`, and so on, to form the array `C`. All input arrays must have the same number of columns.

The class of object arrays `A1,A2,...` must be derived from the same root class of a `matlab.mixin.Heterogeneous` hierarchy.

MATLAB calls:

```
C = vertcat(A1,A2,...)
```

for the expression:

```
C = [A1;A2;...]
```

when `A1` and `A2`, and so on are arrays of `matlab.mixin.Heterogeneous` objects.

If all input arguments are of the same specific class, the class of the resulting array is unchanged. If all input arguments are of different subclasses of a common superclass that is derived from `matlab.mixin.Heterogeneous`, then the result is a heterogeneous array. The array's class is that of the most specific superclass shared by all input arguments.

If all input arguments are not members of the same heterogeneous hierarchy, MATLAB calls the `convertObjects` method, if defined by the dominant root class (the first argument or the left-most element in the concatenation if no other class is dominant).

The `horzcat` method is sealed in the class `matlab.mixin.Heterogeneous` and, therefore, you cannot override it in subclasses.

Input Arguments

A1

Object array of class `matlab.mixin.Heterogeneous`

A2

Object array of class `matlab.mixin.Heterogeneous`

Output Arguments

C

Array resulting from the specified vertical concatenation. The class of this array is that of the most specific superclass shared by the input arguments.

Attributes

`Sealed`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

See Also

`matlab.mixin.Heterogeneous` | `vertcat`

hex2dec

Purpose

Convert hexadecimal number string to decimal number

Syntax

```
d = hex2dec('hex_value')
```

Description

`d = hex2dec('hex_value')` converts *`hex_value`* to its floating-point integer representation. The argument *`hex_value`* is a hexadecimal integer stored in a MATLAB string. The value of *`hex_value`* must be smaller than hexadecimal `10,000,000,000,000`.

If *`hex_value`* is a character array, each row is interpreted as a hexadecimal string.

Examples

```
hex2dec('3ff')

ans =

        1023
```

For a character array `S`,

```
S =
0FF
2DE
123
hex2dec(S)

ans =

255
734
291
```

See Also

`dec2hex` | `format` | `hex2num` | `sprintf`

Purpose

Convert hexadecimal number string to double-precision number

Syntax

```
n = hex2num(S)
```

Description

`n = hex2num(S)`, where `S` is a 16 character string representing a hexadecimal number, returns the IEEE double-precision floating-point number `n` that it represents. Fewer than 16 characters are padded on the right with zeros. If `S` is a character array, each row is interpreted as a double-precision number.

`NaN`s, infinities and denorms are handled correctly.

Examples

```
hex2num('400921fb54442d18')
```

returns `Pi`.

```
hex2num('bff')
```

returns

```
ans =

    -1
```

See Also

`num2hex` | `hex2dec` | `sprintf` | `format`

hgexport

Purpose Export figure

Syntax

```
hgexport(h,filename)
hgexport(h,'-clipboard')
```

Description

`hgexport(h,filename)` writes figure `h` to the file `filename`.

`hgexport(h,'-clipboard')` writes figure `h` to the Microsoft Windows clipboard.

The format in which the figure is exported is determined by which renderer you use. The Painters renderer generates a metafile. The ZBuffer and OpenGL renderers generate a bitmap.

Alternatives

Use the **File > Export Setup** dialog. Use **Edit > Copy Figure** to copy the figure's content to the system clipboard. Fore details, see How to Print or Export.

See Also

`print`

Purpose

Create hggroup object

Syntax

```
h = hggroup
h = hggroup(...,'PropertyName',propertyvalue,...)
```

Properties

For a list of properties, see `Hggroup Properties`.

Description

`h = hggroup` creates an hggroup object as a child of the current axes and returns its handle, `h`.

`h = hggroup(...,'PropertyName',propertyvalue,...)` creates an hggroup object with the property values specified in the argument list. For a description of the properties, see `Hggroup Properties`.

An hggroup object can be the parent of any axes children except light objects, as well as other hggroup objects. You can use hggroup objects to form a group of objects that can be treated as a single object with respect to the following cases:

- Visible — Setting the hggroup object's `Visible` property also sets each child object's `Visible` property to the same value.
- Selectable — Setting each hggroup child object's `HitTest` property to `off` enables you to select all children by clicking any child object.
- Current object — Setting each hggroup child object's `HitTest` property to `off` enables the hggroup object to become the current object when any child object is picked. See the next section for an example.

Examples

This example defines a callback for the `ButtonDownFcn` property of an hggroup object. In order for the hggroup to receive the mouse button down event that executes the `ButtonDownFcn` callback, the `HitTest` properties of all the line objects must be set to `off`. The event is then passed up the hierarchy to the hggroup.

The following function creates a random set of lines that are parented to an hggroup object. The local function `set_lines` defines a callback that executes when the mouse button is pressed over any of the lines.

The callback simply increases the widths of all the lines by 1 with each button press.

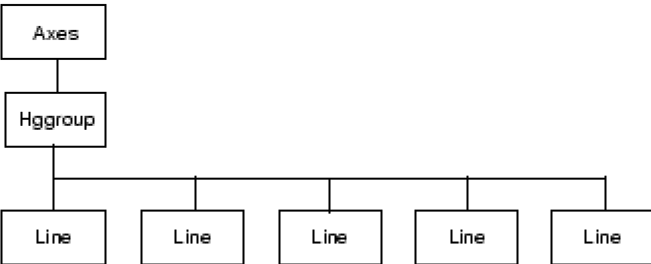
Note If you are using the MATLAB help browser, you can run this example or open it in the MATLAB editor.

```
function doc_hggroup
hg = hggroup('ButtonDownFcn',@set_lines);
hl  = line(randn(5),randn(5),'HitTest','off','Parent',hg);

function set_lines(cb,eventdata)
hl = get(cb,'Children');% cb is handle of hggroup object
lw = get(hl,'LineWidth');% get current line widths
set(hl,{'LineWidth'},num2cell([lw{:}]+1,[5,1])')
```

Note that selecting any one of the lines selects all the lines. (To select an object, enable plot edit mode by selecting **Plot Edit** from the **Tools** menu.)

Instance Diagram for This Example

The following diagram shows the object hierarchy created by this example.



Setting Default Properties

You can set default hggroup properties on the axes, figure, and `root object` levels:

```
set(0,'DefaultHggroupProperty',PropertyValue...)
set(gcf,'DefaultHggroupProperty',PropertyValue...)
set(gca,'DefaultHggroupProperty',PropertyValue...)
```

where *`Property`* is the name of the `hggroup` property whose default value you want to set and `PropertyValue` is the value you are specifying. Use `set` and `get` to access the hggroup properties.

See Also

`hgtransform` | `Hggroup Properties`

How To

- "Group Objects"
- "Function Handle Callbacks"

Hggroup Properties

Purpose

Hggroup properties

Creating Hggroup Objects

Use `hggroup` to create hggroup objects.

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands.

To change the default values of properties, see "Setting Default Property Values".

See "Group Objects" for general information on this type of object.

Hggroup Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`Annotation`

`hg.Annotation` object (read-only)

Control the display of hggroup objects in legends. Specifies whether this hggroup object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the hggroup object is displayed in a figure legend.

IconDisplayStyle Value	Purpose
`on`	Include the hggroup object in a legend as one entry, but not its children objects
`off`	Do not include the hggroup or its children in a legend (default)
`children`	Include only the children of the hggroup as separate entries in the legend

Setting the IconDisplayStyle Property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the IconDisplayStyle Property

See "Controlling Legends" for more information and examples.

`BeingDeleted`

`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`
function handle | cell array containing function handle and additional arguments | string (not recommended)

Button press callback function. Executes whenever you press a mouse button while the pointer is over the children of the hggroup object. Define the `ButtonDownFcn` as a function handle. The function must define at least two input arguments (handle of figure associated with the mouse button press and an empty event structure).

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`Children`

array of graphics object handles

Children of the hggroup object. An array containing the handles of all objects parented to the hggroup object (whether visible or not).

Note that if a child object's `HandleVisibility` property is `callback` or `off`, its handle does not appear in the hggroup `Children` property unless you set the Root `ShowHiddenHandles` property to `on`:

```
set(0,'ShowHiddenHandles','on')
```

`Clipping`

`{on} | off`

Clipping mode. MATLAB clips stairs plots to the axes plot box by default. If you set `Clipping` to `off`, lines might be displayed outside the axes plot box.

`CreateFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback executed during object creation. Executes when MATLAB creates an hggroup object. You must define this property as a default value for hggroup objects or in a call to the `hggroup` function to create a new hggroup object. For example, the statement:

```
set(0,'DefaulthggroupCreateFcn',@myCreateFcn)
```

defines a default value on the root level that applies to every hggroup object created in that MATLAB session. Whenever you create an hggroup object, the function associated with the function handle `@myCreateFcn` executes.

MATLAB executes the callback after setting all the hggroup object's properties. Setting the `CreateFcn` property on an existing hggroup object has no effect.

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DeleteFcn`
function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback executed during object deletion. Executes when the hggroup object is deleted (for example, this might happen when you issue a `delete` command on the hggroup object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`
string

String used by legend. The `legend` function uses the `DisplayName` property to label the hggroup object in the legend. The default is an empty string.

- If you specify string arguments with the `legend` function, MATLAB set `DisplayName` to the corresponding string and uses that string for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, MATLAB set `DisplayName` to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add a legend programmatically that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more information and examples.

`EraseMode`
`{normal} | none | xor | background`

Erase mode. This property controls the technique MATLAB uses to draw and erase hggroup child objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.

- none — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with EraseMode none, you cannot print these objects because MATLAB stores no information about their former locations.
- xor — Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes Color property is none). That is, it isn't erased correctly if there are objects behind it.
- background — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes Color property is none). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

Set the axes background color with the axes Color property. Set the figure background color with the figure Color property.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the EraseMode of all objects is normal. This means graphics objects created with EraseMode set to none, xor, or background can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors (for example, performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the getframe command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing the hggroup object.

- `on` — Handles are always visible.
- `callback` — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties). See also `findall`.

Handle Validity

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

`HitTest`
`{on} | off`

Selectable by mouse click. `HitTest` determines whether the hggroup object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the hggroup child objects. Note that to pick the hggroup object, its children must have their `HitTest` property set to `off`.

If the hggroup object's `HitTest` is `off`, clicking it picks the object behind it.

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the `Interruptible` property affects only the callbacks for the`ButtonDownFcn` property. A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both the callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see "Control Callback Execution and Interruption".

`Parent`
axes handle

Parent of hggroup object. This property contains the handle of the hggroup object's parent object. The parent of an hggroup object is the axes, hggroup, or hgtransform object that contains it.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection handles at the corners and midpoints of hggroup child objects if the `SelectionHighlight` property is also `on` (the default).

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing selection handles on the hggroup child objects. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`Tag`
string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

For example, create an hggroup object and set the `Tag` property:

```
t = hggroup('Tag','group1')
```

When you want to access the object, use `findobj` to find its handle. For example:

```
h = findobj('Tag','group1');
```

`Type`
string (read-only)

Type of graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For hggroup objects, `Type` is '`hggroup`'. The following statement finds all the hggroup objects in the current axes.

```
t = findobj(gca,'Type','hggroup');
```

`UIContextMenu`
handle of uicontextmenu object

Associate a context menu with the hggroup object. Assign this property the handle of a uicontextmenu object created in the hggroup object's figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click the hggroup object.

`UserData`
array

User-specified data. Data you want to associate with the hggroup object (including cell arrays and structures). The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`
`{on} | off`

Visibility of hggroup object and its children. By default, hggroup object visibility is `on`. This means all children of the hggroup are visible unless the child object's `Visible` property is `off`. Setting an hggroup object's `Visible` property to `off` also makes its children invisible.

See Also hggroup

Purpose

Load Handle Graphics object hierarchy from file

Syntax

```
h = hgload('filename')
[h,old_prop_values] = hgload(...,property_structure)
hgload(...,'all')
```

Description

`h = hgload('filename')` loads Handle Graphics objects and its children (if any) from the FIG-file specified by `filename` and returns handles to the top-level objects. If `filename` contains no extension, then the MATLAB software adds the `.fig` extension.

`[h,old_prop_values] = hgload(...,property_structure)` overrides the properties on the top-level objects stored in the FIG-file with the values in `property_structure`, and returns their previous values in `old_prop_values`.

`property_structure` must be a structure having field names that correspond to property names and values that are the new property values.

`old_prop_values` is a cell array equal in length to `h`, containing the old values of the overridden properties for each object. Each cell contains a structure having field names that are property names, each of which contains the original value of each property that has been changed. Any property specified in `property_structure` that is not a property of a top-level object in the FIG-file is not included in `old_prop_values`.

`hgload(...,'all')` overrides the default behavior, which does not reload nonserializable objects saved in the file. These objects include the default toolbars and default menus. This option is obsolete and will be removed in a future release.

Nonserializable objects (such as the default toolbars and the default menus) are normally not reloaded because they are loaded from different files at figure creation time. This allows revisions of the default menus and toolbars to occur without affecting existing FIG-files. Passing the string `all` to `hgload` ensures that any nonserializable objects contained in the file are also reloaded.

Note that, by default, `hgsave` excludes nonserializable objects from the FIG-file unless you use the `all` flag.

Alternatives

Use the **File > Open** on the figure window menu to access figure files with the **Open** dialog.

See Also

`hgsave` | `open`

Purpose

Save Handle Graphics object hierarchy to file

Syntax

```
hgsave('filename')
hgsave(h,'filename')
hgsave(...,'all')
hgsave(...,'-v6')
hgsave(...,'-v7.3')
```

Description

`hgsave('filename')` saves the current figure to a file named `filename`.

`hgsave(h,'filename')` saves the objects identified by the array of handles `h` to a file named `filename`. If you do not specify an extension for `filename`, then the extension `.fig` is appended. If `h` is a vector, none of the handles in `h` may be ancestors or descendents of any other handles in `h`.

`hgsave(...,'all')` overrides the default behavior, which does not save nonserializable objects. Nonserializable objects include the default toolbars and default menus. This allows revisions of the default menus and toolbars to occur without affecting existing FIG-files and also reduces the size of FIG-files. Passing the string `all` to `hgsave` ensures that nonserializable objects are also saved. This option is obsolete and will be removed in a future release.

The default behavior of `hgload` is to ignore nonserializable objects in the file at load time. You can override this behavior using the `all` argument with `hgload`.

`hgsave(...,'-v6')` saves the FIG-file in a format that can be loaded by versions prior to MATLAB 7.

`hgsave(...,'-v7.3')` saves the FIG-file in a format that can be loaded only by MATLAB versions 7.3 and above. This format, based on HDF5 files, is intended for saving FIG-files larger than 2 GB.

You can make `-v6` or `-v7.3` your default format for saving MAT-files and FIG-files by setting a preference, which will eliminate the need to specify the flag each time you save. See "MAT-Files Preferences" in the MATLAB Desktop Tools and Development Environment documentation.

Full Backward Compatibility

When creating a figure you want to save and use in a MATLAB version prior to MATLAB 7, use the `'v6'` option with the plotting function and the `'-v6'` option for `hgsave`. Check the reference page for the plotting function you are using for more information.

See “Plot Objects and Backward Compatibility” for more information.

Alternatives

Use the **File > Export Setup** dialog. Use **Edit > Copy Figure** to copy the figure’s content to the system clipboard. Fore details, see How to Print or Export.

See Also

`hgload` | `open` | `save`

Purpose

Abstract class used to derive handle class with set and get methods

Syntax

`classdef myclass < hgsetget`

Description

`classdef myclass < hgsetget` makes *myclass* a subclass of the `hgsetget` class, which is a subclass of the `handle` class.

Use the `hgsetget` class to derive classes that inherit `set` and `get` methods that behave like Handle Graphics `set` and `get` functions.

Methods

When you derive a class from the `hgsetget` class, your class inherits the following methods.

Method	Purpose
`set`	Assigns values to the specified properties or returns a cell array of possible values for writable properties.
`get`	Returns value of specified property or a `struct` with all property values.
`setdisp`	Called when `set` is called with no output arguments and a handle array, but no property name. Override this method to change what set displays.
`getdisp`	Called when `get` is called with no output arguments and handle array, but no property name. Override this method to change what get displays.

See Also

`handle` | `set (hgsetget)` | `get (hgsetget)` | `set` | `get`

How To

- "Implementing a Set/Get Interface for Properties"

hgtransform

Purpose

Create hgtransform graphics object

Syntax

```
h = hgtransform
h = hgtransform('PropertyName',propertyvalue,...)
```

Properties

For a list of properties, see `Hgtransform Properties`.

Description

`h = hgtransform` creates an hgtransform object and returns its handle.

`h = hgtransform('`*`PropertyName`*`',propertyvalue,...)` creates an hgtransform object with the property value settings specified in the argument list. For a description of the properties, see `Hgtransform Properties`.

hgtransform objects can contain other objects, which lets you treat the hgtransform and its children as a single entity with respect to visibility, size, orientation, etc. You can group objects by parenting them to a single hgtransform object (i.e., setting the object's `Parent` property to the hgtransform object's handle):

```
h = hgtransform;
surface('Parent',h,...)
```

The primary advantage of parenting objects to an hgtransform object is that you can perform *transforms* (for example, translation, scaling, rotation, etc.) on the child objects in unison.

The parent of an hgtransform object is either an axes object or another hgtransform.

Although you cannot see an hgtransform object, setting its `Visible` property to `off` makes all its children invisible as well.

Exceptions and Limitations

- An hgtransform object can be the parent of any number of axes child objects belonging to the same axes, except for light objects.
- hgtransform objects can never be the parent of axes objects and therefore can contain objects only from a single axes.

- hgtransform objects can be the parent of other hgtransform objects within the same axes.
- You cannot transform image objects because images are not true 3-D objects. Texture mapping the image data to a surface `CData` enables you to produce the effect of transforming an image in 3-D space.

Note Many plotting functions clear the axes (i.e., remove axes children) before drawing the graph. Clearing the axes also deletes any hgtransform objects in the axes.

Examples

Transforming a Group of Objects

This example shows how to create a 3-D star with a group of surface objects parented to a single hgtransform object. The hgtransform then rotates the object about the *z*-axis while scaling its size.

Tip If you are using the MATLAB Help browser, you can run this example or open it in the MATLAB Editor.

---

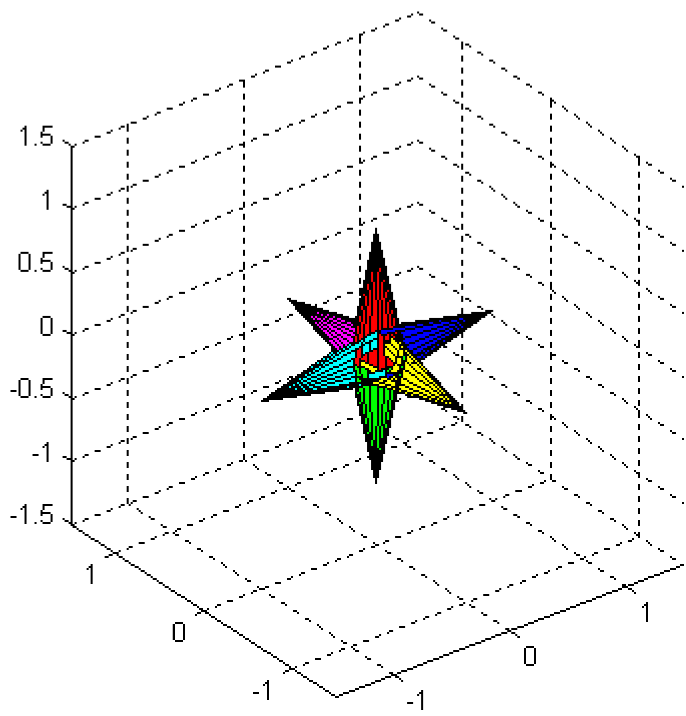
1 Create an axes and adjust the view. Set the axes limits to prevent auto limit selection during scaling.

```
ax = axes('XLim',[-1.5 1.5],'YLim',[-1.5 1.5],...
   'ZLim',[-1.5 1.5]);
view(3); grid on; axis equal
```

2 Create the objects you want to parent to the hgtransform object.

```
[x y z] = cylinder([.2 0]);
h(1) = surface(x,y,z,'FaceColor','red');
h(2) = surface(x,y,-z,'FaceColor','green');
h(3) = surface(z,x,y,'FaceColor','blue');
h(4) = surface(-z,x,y,'FaceColor','cyan');
```

```
h(5) = surface(y,z,x,'FaceColor','magenta');
h(6) = surface(y,-z,x,'FaceColor','yellow');
```

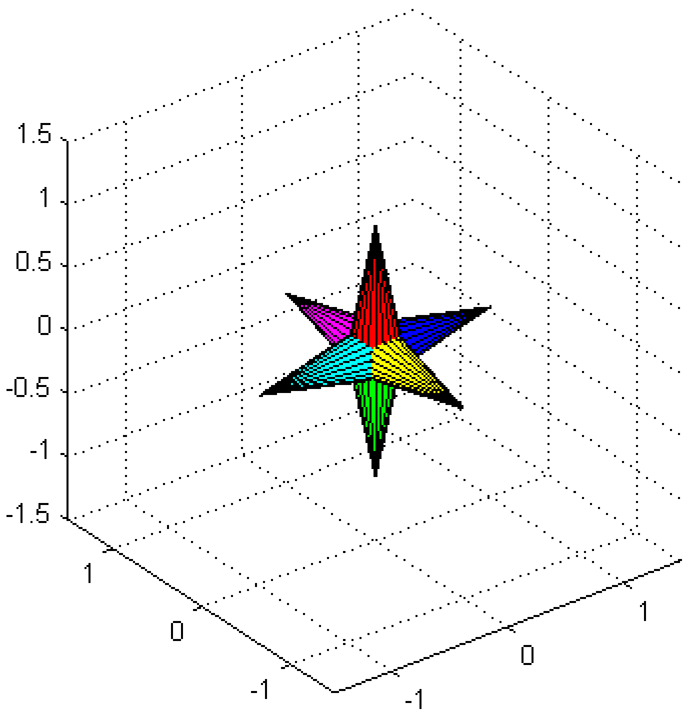


3 Create an hgtransform object and parent the surface objects to it. The figure should not change from the image above.

```
t = hgtransform('Parent',ax);
set(h,'Parent',t)
```

4 Select a renderer and show the objects.

```
set(gcf,'Renderer','opengl')
drawnow
```



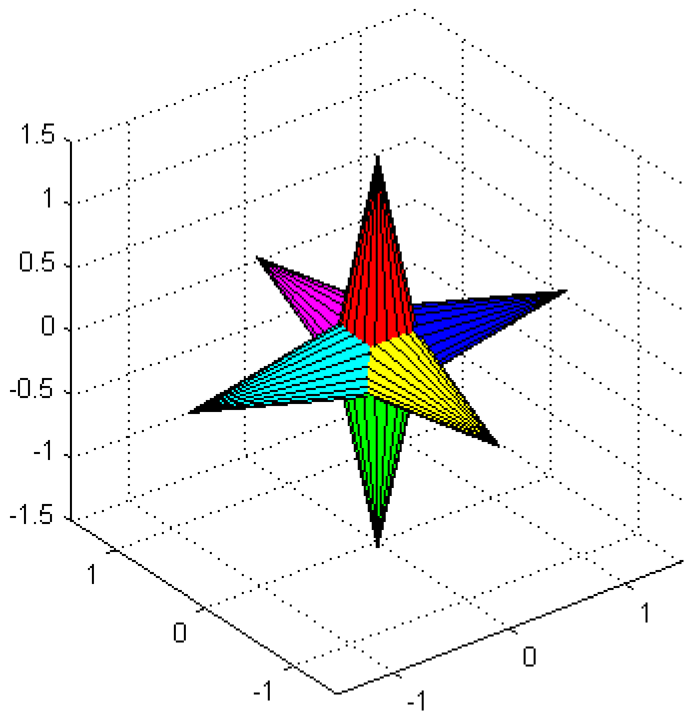
5 Initialize the rotation and scaling matrix to the identity matrix (`eye`). Again, the image should not change.

```
Rz = eye(4);
Sxy = Rz;
```

6 Form the *z*-axis rotation matrix and the scaling matrix. Rotate group and scale by using the increasing values of `r`.

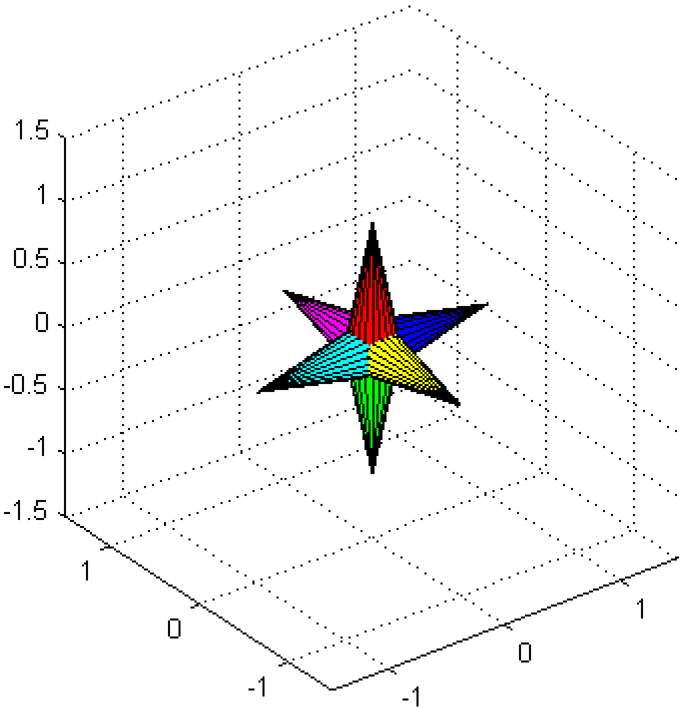
```
for r = 1:.1:2*pi
 % Z-axis rotation matrix
Rz = makehgtform('zrotate',r);
 % Scaling matrix
 Sxy = makehgtform('scale',r/4);
```

```
    % Concatenate the transforms and
    % set the hgtransform Matrix property
        set(t,'Matrix',Rz*Sxy)
        drawnow
end
pause(1)
```



7 Reset to the original orientation and size using the identity matrix.

```
set(t,'Matrix',eye(4))
```



Transforming Objects Independently

This example creates two hgtransform objects to illustrate how to transform each independently within the same axes. A translation transformation moves one hgtransform object away from the origin.

Tip If you are using the MATLAB Help browser, you can run this example or open it in the MATLAB Editor.

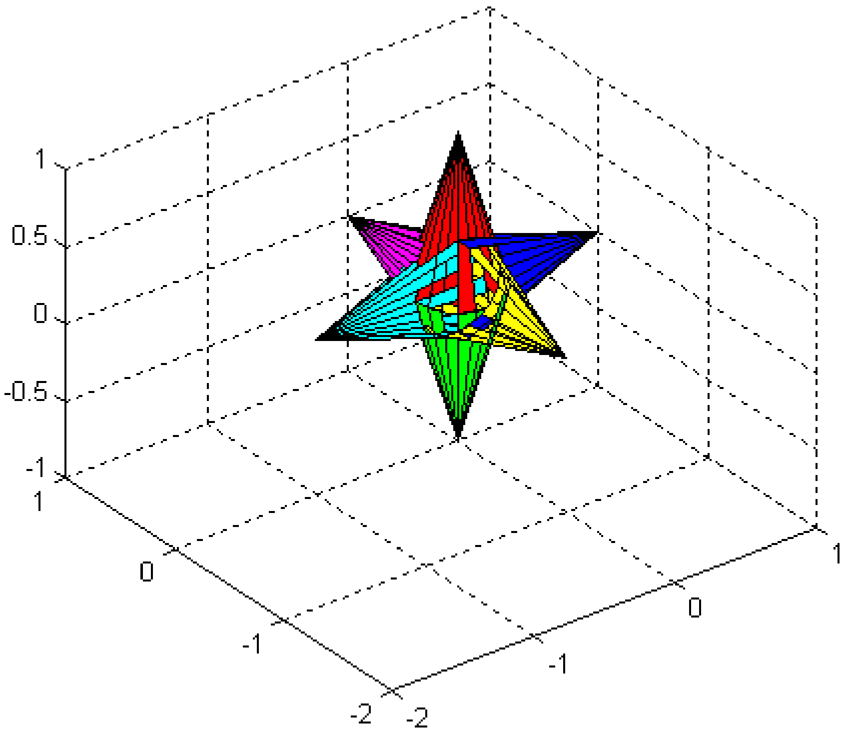
---

1 Create and set up the axes object that will be the parent of both hgtransform objects. Set the limits to accommodate the translated object.

```
ax = axes('XLim',[-2 1],'YLim',[-2 1],'ZLim',[-1 1]);
view(3); grid on; axis equal
```

2 Create the surface objects to group.

```
[x y z] = cylinder([.3 0]);
h(1) = surface(x,y,z,'FaceColor','red');
h(2) = surface(x,y,-z,'FaceColor','green');
h(3) = surface(z,x,y,'FaceColor','blue');
h(4) = surface(-z,x,y,'FaceColor','cyan');
h(5) = surface(y,z,x,'FaceColor','magenta');
h(6) = surface(y,-z,x,'FaceColor','yellow');
```

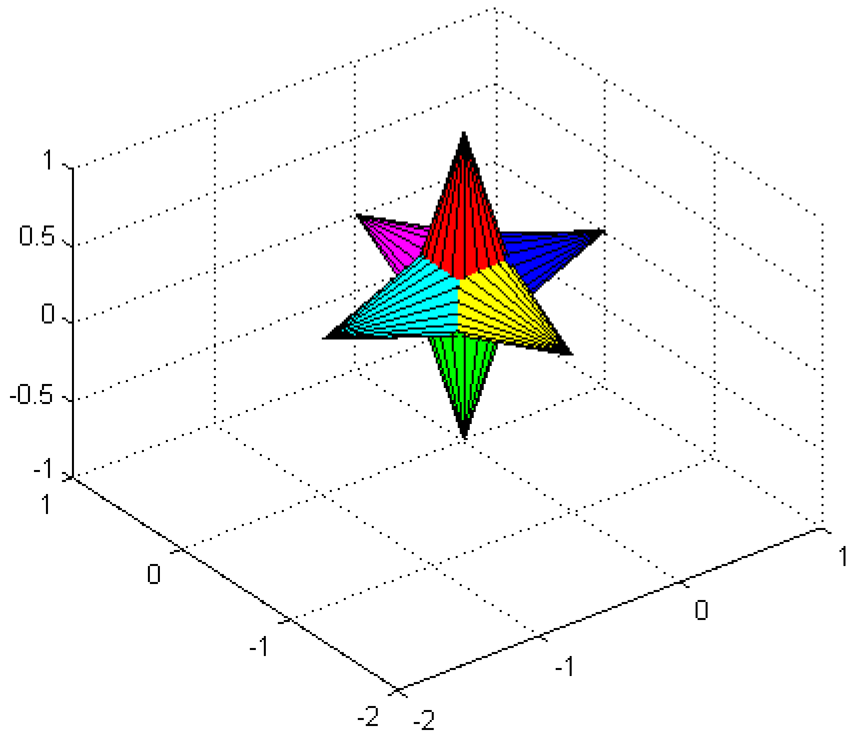


3 Create the hgtransform objects and parent them to the same axes. The figure should not change.

```
t1 = hgtransform('Parent',ax);
t2 = hgtransform('Parent',ax);
```

4 Set the renderer to use OpenGL.

```
set(gcf,'Renderer','opengl')
```

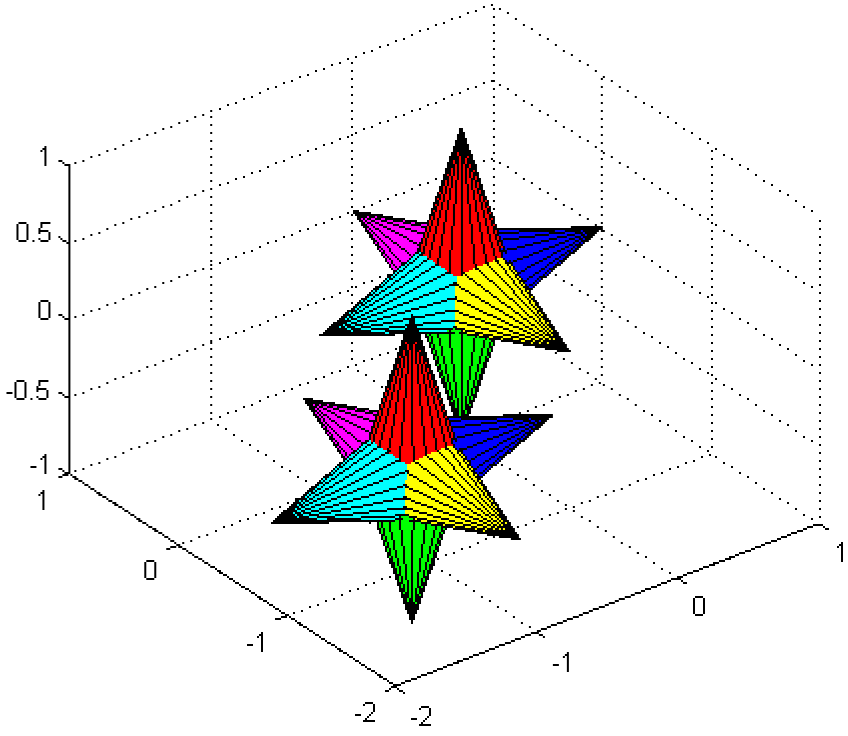


5 Parent the surfaces to hgtransform `t1`, then copy the surface objects and parent the copies to hgtransform `t2`. This figure should not change.

```
set(h,'Parent',t1)
h2 = copyobj(h,t2);
```

6 Translate the second hgtransform object away from the first hgtransform object and display the result.

```
Txy = makehgtform('translate',[-1.5 -1.5 0]);
set(t2,'Matrix',Txy)
drawnow
```



7 Rotate both hgtransform objects in opposite directions. The final image for this step is the same as for step 6. However, you should run the code to see the rotations.

```
% Rotate 10 times (2pi radians = 1 rotation)
for r = 1:.1:20*pi
 % Form z-axis rotation matrix
 Rz = makehgtform('zrotate',r);
 % Set transforms for both hgtransform objects
 set(t1,'Matrix',Rz)
 set(t2,'Matrix',Txy*inv(Rz))
 drawnow
end
```

Setting Default Properties

You can set default hgtransform properties on the axes, figure, and `root object` levels:

```
set(0,'DefaultHgtransformPropertyName',propertyvalue,...)
set(gcf,'DefaultHgtransformPropertyName',propertyvalue,...)
set(gca,'DefaultHgtransformPropertyName',propertyvalue,...)
```

`PropertyName` is the name of the hgtransform property and `propertyvalue` is the specified value. Use `set` and `get` to access hgtransform properties.

See Also

`hggroup` | `makehgtform` | `Hgtransform Properties`

How To

- “Group Objects”

Hgtransform Properties

Purpose

Hgtransform properties

Creating Hgtransform Objects

Use `hgtransform` to create hgtransform objects.

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands.

To change the default values of properties, see "Setting Default Property Values".

See "Group Objects" for general information on this type of object.

Hgtransform Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`Annotation`

`hg.Annotation` object (read-only)

Control the display of hgtransform objects in legends. Specifies whether this hgtransform object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the hgtransform object is displayed in a figure legend.

IconDisplayStyle Value	Purpose
`on`	Include the hgtransform object in a legend as one entry, but not its children objects
`off`	Do not include the hgtransform or its children in a legend (default)
`children`	Include only the children of the hgtransform as separate entries in the legend

Setting the IconDisplayStyle Property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the IconDisplayStyle Property

See "Controlling Legends" for more information and examples.

`BeingDeleted`

`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Button press callback function. Executes whenever you press a mouse button while the pointer is within the extent of the hgtransform object, but not over another graphics object. The extent of an hgtransform object is the smallest rectangle that encloses all the children. Note that you cannot execute the hgtransform object's button down function if it has no children.

Define the `ButtonDownFcn` as a function handle. The function must define at least two input arguments (handle of figure

associated with the mouse button press and an empty event structure).

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`Children`

array of graphics object handles

Children of the hgtransform object. An array containing the handles of all graphics objects parented to the hgtransform object (whether visible or not).

The graphics objects that can be children of an hgtransform are images, lights, lines, patches, rectangles, surfaces, and text. You can change the order of the handles and thereby change the stacking of the objects on the display.

Note that if a child object's `HandleVisibility` property is `callback` or `off`, its handle does not show up in the hgtransform `Children` property unless you set the Root `ShowHiddenHandles` property to `on`.

`Clipping`

`{on} | off`

Clipping mode. This property has no effect on hgtransform objects.

`CreateFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback executed during object creation. Executes when MATLAB creates an hgtransform object. You must define this property as a default value for hgtransform objects. For example, the statement:

```
set(0,'DefaultHgtransformCreateFcn',@myCreateFcn)
```

defines a default value on the root level that applies to every hgtransform object created in a MATLAB session. Whenever you create an hgtransform object, the function associated with the function handle `@myCreateFcn` executes.

MATLAB executes the callback after setting all the hgtransform object's properties. Setting the `CreateFcn` property on an existing hgtransform object has no effect.

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`DeleteFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback executed during object deletion. Executes when the hgtransform object is deleted (for example, this might happen when you issue a `delete` command on the hgtransform object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values.

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is accessible through the root `CallbackObject` property, which can be queried using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`
string

String used by legend. The `legend` function uses the `DisplayName` property to label the hgtransform object in the legend. The default is an empty string.

- If you specify string arguments with the `legend` function, MATLAB set `DisplayName` to the corresponding string and uses that string for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, MATLAB set `DisplayName` to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add a legend programmatically that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more information and examples.

`EraseMode`
`{normal} | none | xor | background`

Erase mode. This property controls the technique MATLAB uses to draw and erase hgtransform child objects (light objects have no erase mode). Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all

objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.

- `none` — Do not erase objects when they are moved or destroyed. While the objects are still visible on the screen after erasing with `EraseMode none`, you cannot print these objects because MATLAB stores no information about their former locations.
- `xor`— Draw and erase the object by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the object does not damage the color of the objects behind it. However, the color of the erased object depends on the color of the screen behind it and it is correctly colored only when it is over the axes background color (or the figure background color if the axes `Color` property is `none`). That is, it isn't erased correctly if there are objects behind it.
- `background` — Erase the graphics objects by redrawing them in the axes background color, (or the figure background color if the axes `Color` property is `none`). This damages other graphics objects that are behind the erased object, but the erased object is always properly colored.

  Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors (for example, performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing the hgtransform object.

- `on` — Handles are always visible.
- `callback` — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in

the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties). See also `findall`.

Handle Validity

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

`HitTest`
`{on} | off`

Selectable by mouse click. `HitTest` determines whether the hgtransform object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click within the limits of the hgtransform object. If `HitTest` is `off`, clicking the hgtransform picks the object behind it.

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the `Interruptible` property affects only the callbacks for the`ButtonDownFcn` property. A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB

handles both the callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see "Control Callback Execution and Interruption".

`Matrix`
4-by-4 matrix

Transformation matrix applied to hgtransform object and its children. The hgtransform object applies the transformation matrix to all its children.

See "Group Objects" for more information and examples.

`Parent`
figure handle

Parent of hgtransform object. This property contains the handle of the hgtransform object's parent object. The parent of an hgtransform object is the axes, hggroup, or hgtransform object that contains it.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When you set this property to `on`, MATLAB displays selection handles on all child objects of the hgtransform if the `SelectionHighlight` property is also `on` (the default).

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing selection handles on the objects parented to the hgtransform. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`Tag`
string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

For example, create an hgtransform object and set the `Tag` property:

```
t = hgtransform('Tag','subgroup1')
```

When you want to access the hgtransform object to add another object, use `findobj` to find the hgtransform object's handle. The following statement adds a line to `subgroup1` (assuming `x` and `y` are defined).

```
line('XData',x,'YData',y,'Parent',findobj('Tag','subgroup1'))
```

`Type`
string (read-only)

Type of graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For hgtransform objects, `Type` is '`hgtransform`'. The following statement finds all the hgtransform objects in the current axes.

```
t = findobj(gca,'Type','hgtransform');
```

`UIContextMenu`
handle of uicontextmenu object

Associate a context menu with the hgtransform object. Assign this property the handle of a uicontextmenu object created in the hgtransform object's figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the extent of the hgtransform object.

`UserData`

array

User-specified data. Data you want to associate with the hgtransform object (including cell arrays and structures). The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`

`{on} | off`

Visibility of hgtransform object and its children. By default, hgtransform object visibility is `on`. This means all children of the hgtransform are visible unless the child object's `Visible` property is `off`. Setting an hgtransform object's `Visible` property to `off` also makes its children invisible.

See Also

`hgtransform`

Purpose

Remove hidden lines from mesh plot

Syntax

```
hidden on
hidden off
hidden
```

Description

Hidden line removal draws only those lines that are not obscured by other objects in a 3-D view. The `hidden` function only applies to surface plot objects that have a uniform `FaceColor`.

`hidden on` turns on hidden line removal for the current mesh plot so lines in the back of a mesh are hidden by those in front. This is the default behavior.

`hidden off` turns off hidden line removal for the current mesh plot.

`hidden` toggles the hidden line removal state.

Algorithms

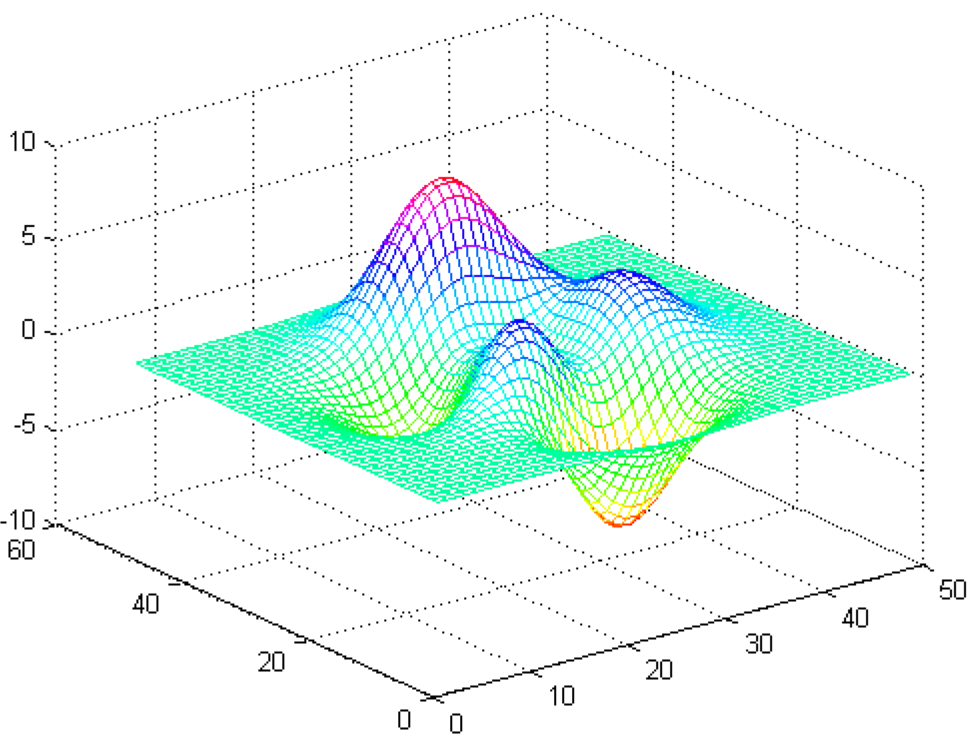
When a surface graphics object has a uniform `FaceColor` matching the `Color` property of the axes, `hidden off` sets the `FaceColor` of the surface object to `'none'`.

`hidden on` sets the `FaceColor` property of such surface objects to match the axes `Color` property (or to match that of the figure, if axes `Color` is `'none'`).

Examples

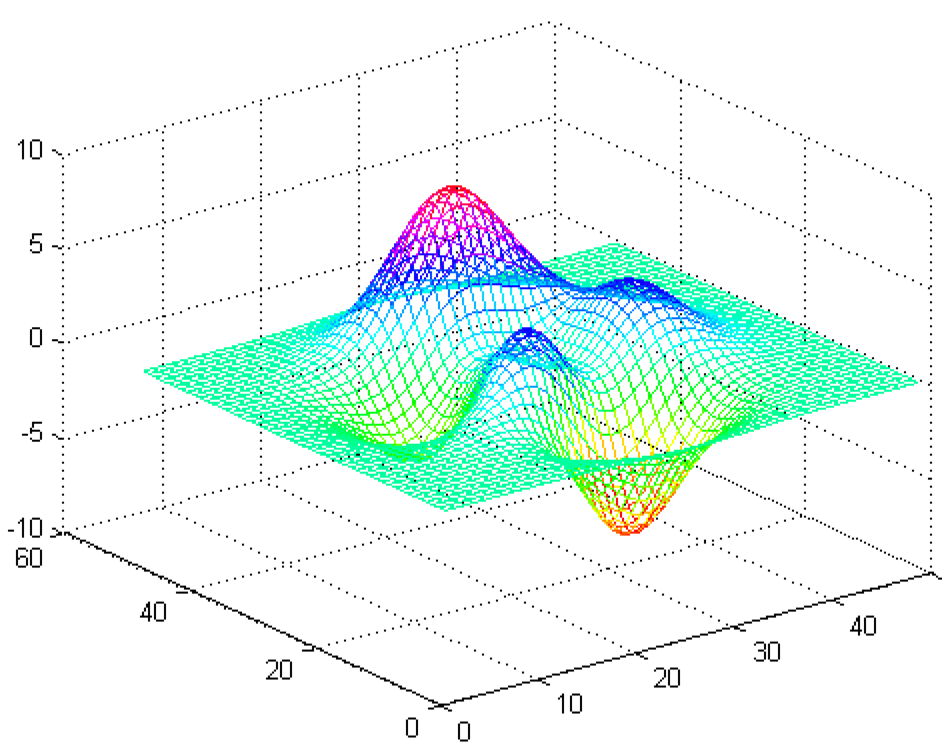
By default hidden lines are removed (`hidden on`) for mesh plots, such as this one of the `peaks` function.

```
mesh(peaks)
colormap hsv
```



Set hidden line removal `off` to show the obscured parts of the mesh.

```
hidden off
```



See Also

`shading` | `mesh` | EdgeColor | FaceColor

hilb

Purpose

Hilbert matrix

Syntax

```
H = hilb(n)
```

Description

`H = hilb(n)` returns the Hilbert matrix of order `n`.

Definitions

The Hilbert matrix is a notable example of a poorly conditioned matrix [1]. The elements of the Hilbert matrices are $H(i,j) = 1/(i + j - 1)$.

Examples

Even the fourth-order Hilbert matrix shows signs of poor conditioning.

```
cond(hilb(4)) =
    1.5514e+04
```

References

[1] Forsythe, G. E. and C. B. Moler, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, 1967, Chapter 19.

See Also

`invhilb`

hist

Purpose

Histogram plot

Syntax

```
hist(data)
hist(data,nbins)
hist(data,xcenters)

hist(axes_handle, ___ )

nelements = hist( ___ )
[nelements,xcenters] = hist( ___ )
```

Description

`hist(data)` creates a histogram bar plot of `data`. Elements in `data` are sorted into 10 equally spaced bins along the *x*-axis between the minimum and maximum values of `data`. Bins are displayed as rectangles such that the height of each rectangle indicates the number of elements in the bin.

`hist(data,nbins)` sorts `data` into the number of bins specified by `nbins`.

`hist(data,xcenters)` sorts `data` into the number of bins determined by `length(xcenters)`. The values in `xcenters` specify the centers for each bin on the *x*-axis.

`hist(axes_handle, ___ )` plots into the axes specified by `axes_handle` instead of into the current axes (`gca`). The option `axes_handle` can precede any of the input argument combinations in the previous syntaxes.

`nelements = hist( ___ )` returns a row vector, `nelements`, indicating the number of elements in each bin.

`[nelements,xcenters] = hist(___)` returns an additional row vector, `xcenters`, indicating the location of each bin center on the *x*-axis. To plot the histogram, you can use `bar(xcenters,nelements)`.

Input Arguments

data - Data to distribute among bins

vector or matrix

Data to distribute among bins, specified as a vector or a matrix.

- If `data` is a vector, then one histogram is created.
- If `data` is a matrix, then a histogram is created separately for each column. Each histogram plot is displayed on the same figure with a different color.

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

nbins - Number of bins

10 (default) | scalar

Number of bins, specified as a scalar.

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

xcenters - Bin centers

vector

Bin centers, specified as a vector. If the values in `xcenters` are not equally spaced, then these values are indicated by markers along the *x*-axis. The edges of the first and last bin can extend to cover the minimum and maximum data values in `data`.

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

axes_handle - Axes handle

Axes handle, which is the reference to an axes object. Use the `gca` function to get the handle to the current axes, for example, `axes_handle = gca;`.

Output Arguments

nelements - Number of elements in each bin

row vector

Number of elements in each bin, returned as a row vector.

xcenters - Bin centers

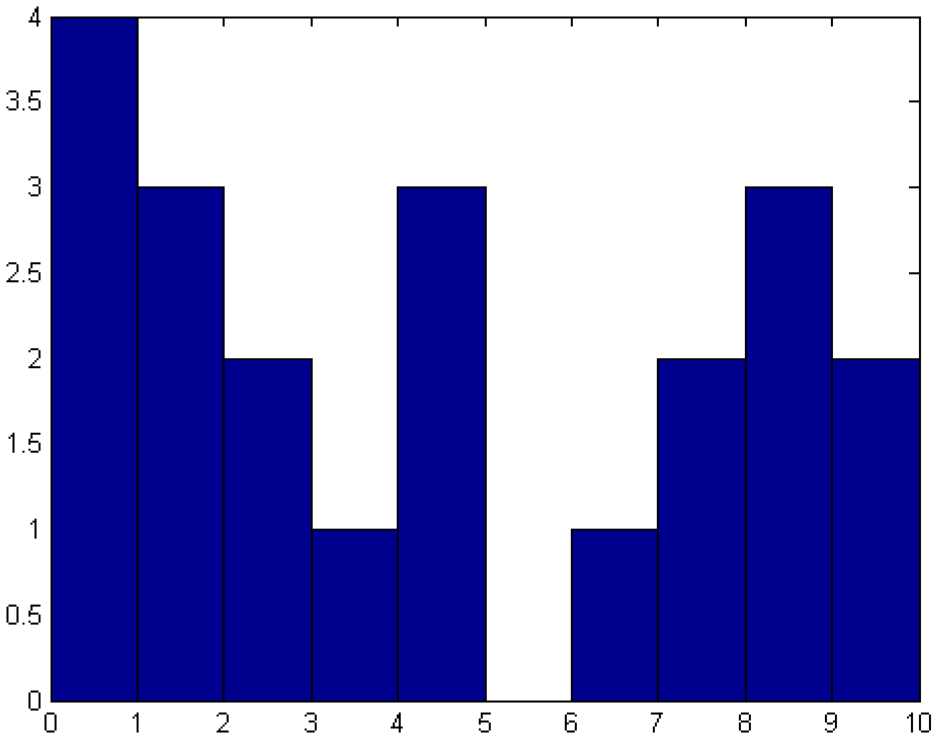
row vector

Bin centers, returned as a row vector. If used with the syntax `[nelements,xcenters] = hist(data,xcenters)`, then the output `xcenters` is equal to the input `xcenters`.

Examples

Create Histogram Bar Plot with Vector Input

```
figure
data = [0,2,9,2,5,8,7,3,1,9,4,3,5,8,10,0,1,2,9,5,10];
hist(data)
```



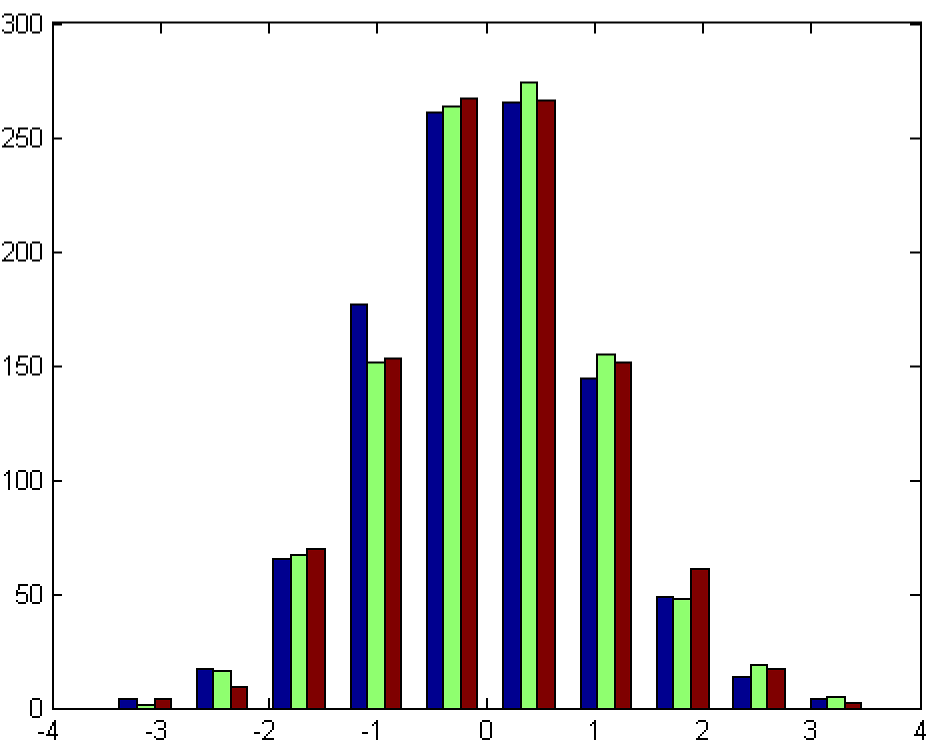
The values in `data` are sorted among 10 equally spaced bins between 0 and 10, the minimum and maximum values.

Create Histogram Bar Plot with Matrix Input

Initialize the random-number generator to make the output of `randn` repeatable. Generate 1,000 normally distributed pseudorandom numbers and create a histogram bar plot.

```
figure
rng(0,'twister')
```

```
data = randn(1000,3);
hist(data)
```



The values in `data` are sorted among 10 equally spaced bins between the minimum and maximum values. Columns of `data` are sorted separately and the histogram for each column is plotted with a different color.

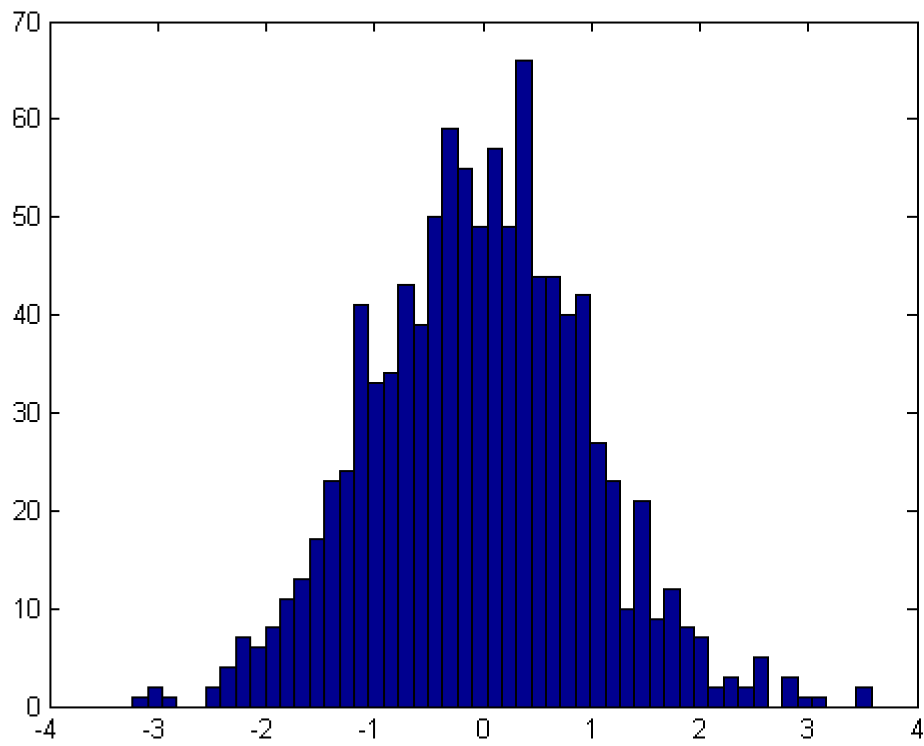
Specify Number of Bins

Initialize the random-number generator to make the output of `randn` repeatable. Generate 1,000 normally distributed pseudorandom numbers.

```
rng(0,'twister')
data = randn(1000,1);
```

Create a histogram plot of `data` sorted into 50 equally spaced bins.

```
figure
nbins = 50;
hist(data,nbins)
```



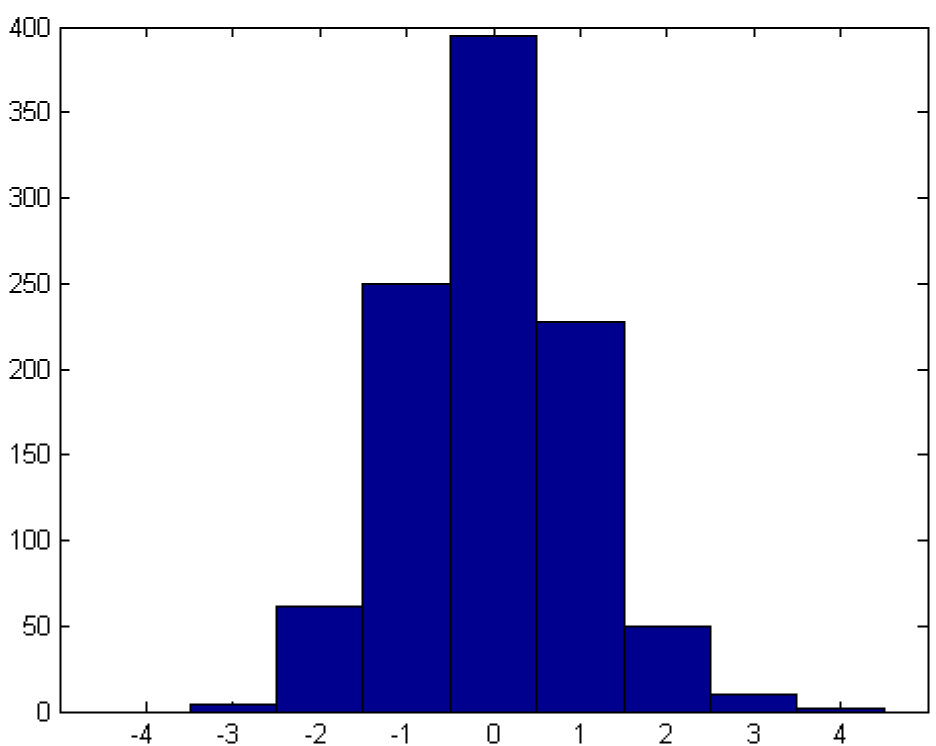
Specify Bin Centers

Initialize the random-number generator to make the output of `randn` repeatable. Generate 1,000 normally distributed pseudorandom numbers.

```
rng(0,'twister')
data = randn(1000,1);
```

Create a histogram plot and specify the center for each bin.

```
figure
xcenters = -4:4;
hist(data,xcenters)
```



Specify Axes for Histogram Bar Plot

Initialize the random-number generator to make the output of `randn` repeatable. Generate 1,000 normally distributed pseudorandom numbers.

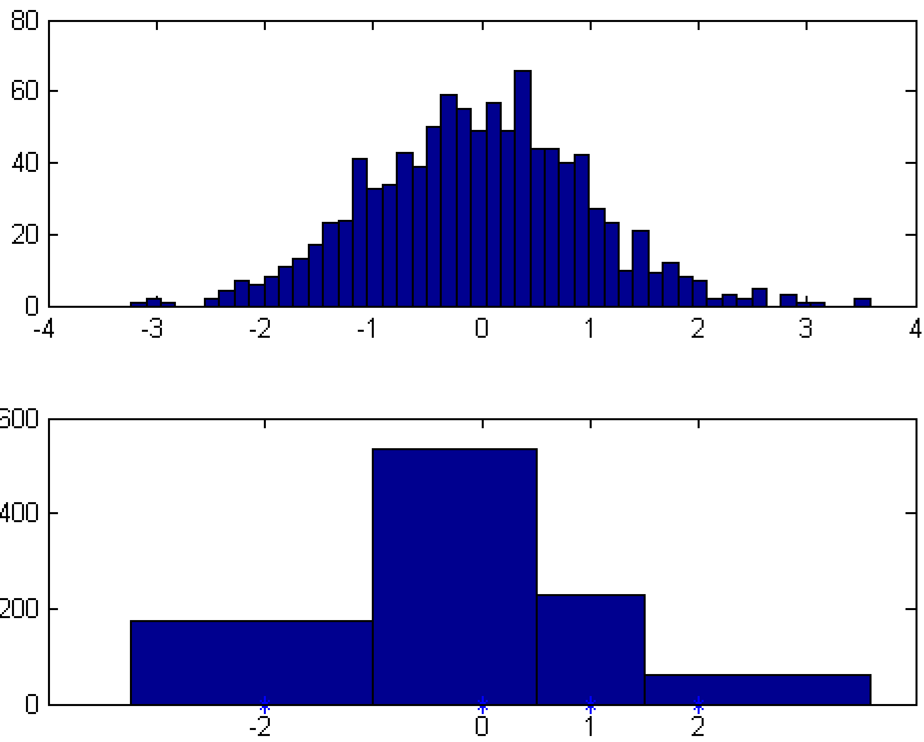
```
rng(0,'twister')
data = randn(1000,1);
```

Set up a vector to specify values for the bin centers.

```
xcenters = [-2,0,1,2];
```

Create a figure with handles to two subplots. In the upper subplot, plot a histogram of `data` sorted into 50 equally spaced bins. In the lower subplot, plot a histogram of the same data and use `xcenters` to specify the bin centers.

```
figure
s(1) = subplot(2,1,1);
s(2) = subplot(2,1,2);
hist(s(1),data,50)
hist(s(2),data,xcenters)
```



In the lower subplot, the specified bin centers are not equally spaced. `hist` indicates the specified values by markers along the *x*-axis.

Plot Histogram Using Bar

Initialize the random-number generator to make the output of `randn` repeatable. Generate 1,000 normally distributed pseudorandom numbers.

```
rng(0,'twister')
data = randn(1000,1);
```

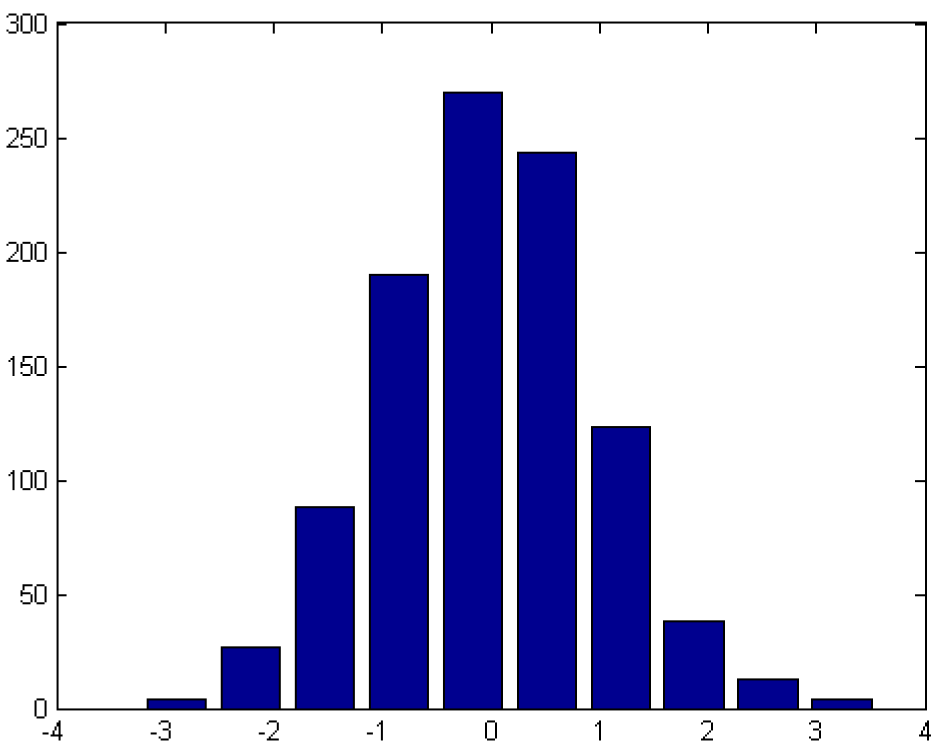
Sort `data` into 10 equally spaced bins. Get the number of elements in each bin and the locations of the bin centers.

```
[nelements,xcenters] = hist(data)
```

```
nelements =
     4    27    88   190   270   243   123    38    13     4
xcenters =
   -2.8915   -2.2105   -1.5294   -0.8484   -0.1673    0.5137    1.1947
```

Use `bar` to plot the histogram.

```
figure
bar(xcenters,nelements)
```



Change Histogram Color Properties

IInitialize the random-number generator to make the output of `randn` repeatable. Generate 1,000 normally distributed pseudorandom numbers and create a histogram bar plot.

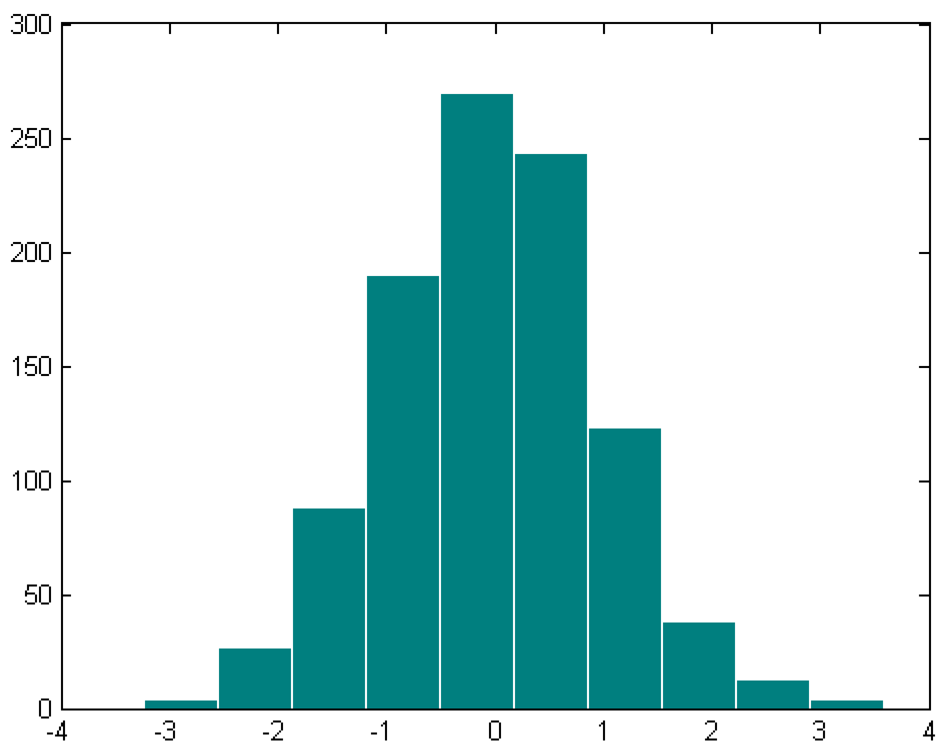
```
figure
rng(0,'twister')
data = randn(1000,1);
hist(data)
```

Get the handle to the patch object that creates the histogram bar plot.

```
h = findobj(gca,'Type','patch');
```

Use the handle to change the face color and the edge color of the bars plotted.

```
set(h,'FaceColor',[0 .5 .5],'EdgeColor','w')
```



Tips

- Use `histc` if it is more natural to specify the bin edges.
- The *x*-axis reflects the range of values in `Y`. The *y*-axis ranges from 0 to the greatest number of elements deposited in any bin.

See Also

`bar` | `ColorSpec` | `histc` | `mode` | `patch` | `rose` | `stairs`

Purpose

Histogram count

Syntax

```
n = histc(x,edges)
n = histc(x,edges,dim)
[n,bin] = histc(...)
```

Description

`n = histc(x,edges)` counts the number of values in vector `x` that fall between the elements in the `edges` vector (which must contain monotonically nondecreasing values). `n` is a `length(edges)` vector containing these counts. No elements of `x` can be complex.

`n(k)` counts the value `x(i)` if `edges(k) <= x(i) < edges(k+1)`. The last bin counts any values of `x` that match `edges(end)`. Values outside the values in `edges` are not counted. Use `-inf` and `inf` in `edges` to include all non-`NaN` values.

For matrices, `histc(x,edges)` returns a matrix of column histogram counts. For N-D arrays, `histc(x,edges)` operates along the first nonsingleton dimension.

`n = histc(x,edges,dim)` operates along the dimension `dim`.

`[n,bin] = histc(...)` also returns an index matrix `bin`. If `x` is a vector, `n(k) = sum(bin==k)`. `bin` is zero for out of range values. If `x` is an `M`-by-`N` matrix, then

```
for j=1:N,
n(k,j) = sum(bin(:,j)==k);
end
```

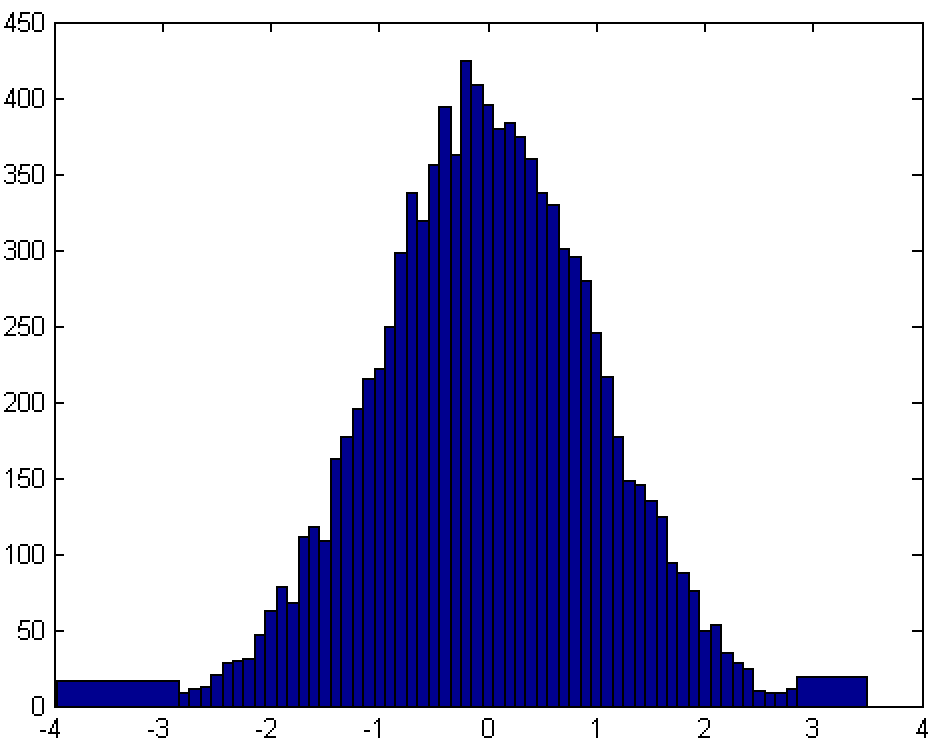
To plot the histogram, use the `bar` command.

Examples

Generate a cumulative histogram of a distribution.

Consider the following distribution:

```
x = -2.9:0.1:2.9;
y = randn(10000,1);
figure(1)
hist(y,x)
```



Calculate number of elements in each bin

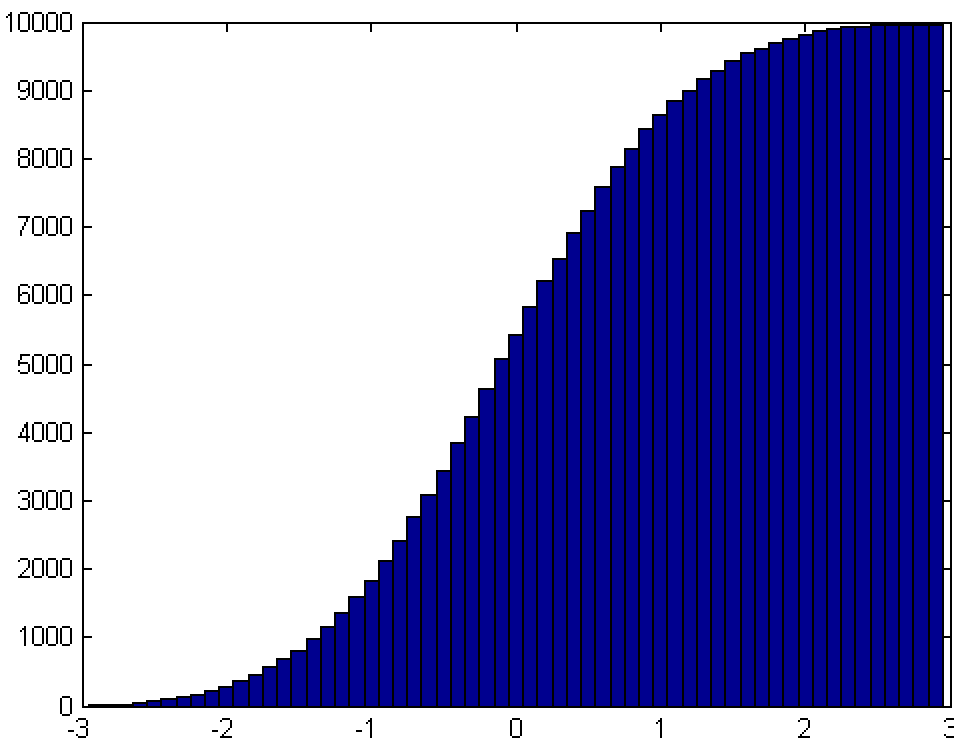
```
n_elements = histc(y,x);
```

Calculate the cumulative sum of these elements using `cumsum`

```
c_elements = cumsum(n_elements)
```

Plot the cumulative distribution like a histogram using `bar`:

```
figure(2)
bar(x,c_elements,'BarWidth',1)
```



See Also

`bar` | `hist` | `mode`

hold

Purpose

Retain current graph when adding new graphs

Syntax

```
hold on
hold off
hold all
hold
hold(axes_handle,...)
```

Description

The `hold` function controls whether MATLAB clears the current graph when you make subsequent calls to plotting functions (the default), or adds a new graph to the current graph.

`hold on` retains the current graph and adds another graph to it. MATLAB adjusts the axes limits, tick marks, and tick labels as necessary to display the full range of the added graph.

`hold off` resets hold state to the default behavior, in which MATLAB clears the existing graph and resets axes properties to their defaults before drawing new plots.

`hold all` holds the graph and the current line color and line style so that subsequent plotting commands do not reset the `ColorOrder` and `LineStyleOrder` property values to the beginning of the list. Plotting commands continue cycling through the predefined colors and line styles from where the last graph stopped in the list.

`hold` reverses the current hold state. If the hold state is currently on, then a `hold` command sets the state to off. Similarly, if the hold state is currently off, then a `hold` command sets the state to on.

`hold(axes_handle,...)` applies the hold to the axes identified by the handle `axes_handle`. If several axes objects exist in a figure window, each axes has its own hold state. `hold` also creates an axes if one does not exist.

Test the hold state using the `ishold` function.

Examples

Create a graph, set `hold` to `on`, then create another graph in the same axes:

```
x = -pi:pi/20:pi;
plot(sin(x))
hold on
plot(cos(x))
hold off
```

If the range of subsequently added data is much greater than the original data, the original graph can become difficult to see in one axes. In these cases, it is usually better to use two separate axes. See `subplot` to create multiple axes in one figure.

Algorithm

`hold` toggles the `NextPlot` axes property between the `add` and `replace`.

`hold on` sets the `NextPlot` property of the current figure and axes to `add`. `hold off` sets the `NextPlot` property of the current axes to `replace`.

See Also

`axis` | `cla` | `ishold` | `newplot`

How To

- `NextPlot`
- “Controlling Graphics Output”

home

Purpose

Send cursor home

Syntax

```
home
```

Description

`home` moves the cursor to the upper-left corner of the window. When using the MATLAB desktop, `home` also scrolls the visible text in the window up and out of view. You can use the scroll bar to see what was previously on the screen.

Examples

Execute a MATLAB command that displays something in the Command Window and then run the `home` function. `home` moves the cursor to the upper-left corner of the screen and clears the screen.

```
magic(5)

ans =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

home
```

See Also

`clc`

Purpose

Concatenate arrays horizontally

Syntax

```
C = horzcat(A1, A2, ...)
```

Description

`C = horzcat(A1, A2, ...)` horizontally concatenates matrices `A1`, `A2`, and so on. All matrices in the argument list must have the same number of rows.

`horzcat` concatenates N-dimensional arrays along the second dimension. The first and remaining dimensions must match.

MATLAB calls `C = horzcat(A1, A2,...)` for the syntax `C = [A1 A2 ...]` when any of `A1`, `A2`, etc., is an object.

Tips

For information on combining unlike integer types, integers with nonintegers, cell arrays with non-cell arrays, or empty matrices with other elements, see "Valid Combinations of Unlike Classes".

Examples

Create a 3-by-5 matrix, `A`, and a 3-by-3 matrix, `B`. Then horizontally concatenate `A` and `B`.

```
A = magic(5);            % Create 3-by-5 matrix, A
A(4:5,:) = []

A =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22


B = magic(3)*100       % Create 3-by-3 matrix, B

B =

   800   100   600
   300   500   700
```

```
   400   900   200

C = horzcat(A, B)      % Horizontally concatenate A and B

C =

    17    24     1     8    15   800   100   600
    23     5     7    14    16   300   500   700
     4     6    13    20    22   400   900   200
```

See Also

`vertcat` | `cat` | `strcat` | `char` | `special character`

How To

- "Redefining Concatenation for Your Class"

Purpose

Horizontal concatenation for `tscollection` objects

Syntax

```
tsc = horzcat(tsc1,tsc2,...)
```

Description

`tsc = horzcat(tsc1,tsc2,...)` performs horizontal concatenation for `tscollection` objects:

```
tsc = [tsc1 tsc2 ...]
```

This operation combines multiple `tscollection` objects, which must have the same time vectors, into one `tscollection` containing `timeseries` objects from all concatenated collections.

See Also

`tscollection` | `vertcat (tscollection)`

hostid

Purpose

Server host identification number

Note `hostid` will be removed in a future release.

Syntax

```
id = hostid
```

Description

`id = hostid` usually returns a single element cell array containing the MATLAB server host identifier as a string. On UNIX[5] platforms, there can be more than one identifier. In that case, `hostid` returns a cell array with an identifier in each cell.

5. UNIX is a registered trademark of The Open Group in the United States and other countries.

Purpose

Convert HSV colormap to RGB colormap

Syntax

```
M = hsv2rgb(H)
rgb_image = hsv2rgb(hsv_image)
```

Description

`M = hsv2rgb(H)` converts a hue-saturation-value (HSV) colormap to a red-green-blue (RGB) colormap. `H` is an *m*-by-3 matrix, where *m* is the number of colors in the colormap. The columns of `H` represent hue, saturation, and value, respectively. `M` is an *m*-by-3 matrix. Its columns are intensities of red, green, and blue, respectively.

`rgb_image = hsv2rgb(hsv_image)` converts the HSV image to the equivalent RGB image. HSV is an *m*-by-*n*-by-3 image array whose three planes contain the hue, saturation, and value components for the image. RGB is returned as an *m*-by-*n*-by-3 image array whose three planes contain the red, green, and blue components for the image.

Tips

As `H(:,1)` varies from 0 to 1, the resulting color varies from red through yellow, green, cyan, blue, and magenta, and returns to red. When `H(:,2)` is 0, the colors are unsaturated (i.e., shades of gray). When `H(:,2)` is 1, the colors are fully saturated (i.e., they contain no white component). As `H(:,3)` varies from 0 to 1, the brightness increases.

The MATLAB `hsv` colormap uses `hsv2rgb([huesaturationvalue])` where `hue` is a linear ramp from 0 to 1, and `saturation` and `value` are all 1's.

See Also

`brighten` | `colormap` | `rgb2hsv`

hypot

Purpose

Square root of sum of squares

Syntax

```
c = hypot(a,b)
```

Description

`c = hypot(a,b)` returns the element-wise result of the following equation, computed to avoid underflow and overflow:

```
c = sqrt(abs(a).^2 + abs(b).^2)
```

Inputs `a` and `b` must follow these rules:

- Both `a` and `b` must be single- or double-precision, floating-point arrays.
- The sizes of the `a` and `b` arrays must either be equal, or one a scalar and the other nonscalar. In the latter case, `hypot` expands the scalar input to match the size of the nonscalar input.
- If `a` or `b` is an empty array (0-by-N or N-by-0), the other must be the same size or a scalar. The result `c` is an empty array having the same size as the empty input(s).

`hypot` returns the following in output `c`, depending upon the types of inputs:

- If the inputs to `hypot` are complex (`w+xi` and `y+zi`), then the statement `c = hypot(w+xi,y+zi)` returns the *positive real* result

  ```
  c = sqrt(abs(w).^2+abs(x).^2+abs(y).^2+abs(z).^2)
  ```

- If `a` or `b` is `–Inf`, `hypot` returns `Inf`.
- If neither `a` nor `b` is `Inf`, but one or both inputs is `NaN`, `hypot` returns `NaN`.
- If all inputs are finite, the result is finite. The one exception is when both inputs are very near the value of the MATLAB constant `realmax`. The reason for this is that the equation `c = hypot(realmax,realmax)` is theoretically `sqrt(2)*realmax`, which overflows to `Inf`.

Examples

Example 1

To illustrate the difference between using the hypot function and coding the basic hypot equation in M-code, create an anonymous function that performs the same function as hypot, but without the consideration to underflow and overflow that hypot offers:

```
myhypot = @(a,b)sqrt(abs(a).^2+abs(b).^2);
```

Find the upper limit at which your coded function returns a useful value. You can see that this test function reaches its maximum at about 1e154, returning an infinite result at that point:

```
myhypot(1e153,1e153)
ans =
    1.4142e+153

myhypot(1e154,1e154)
ans =
   Inf
```

Do the same using the hypot function, and observe that hypot operates on values up to about 1e308, which is approximately equal to the value for realmax on your computer (the largest double-precision floating-point number you can represent on a particular computer):

```
hypot(1e308,1e308)
ans =
  1.4142e+308

hypot(1e309,1e309)
ans =
   Inf
```

Example 2

hypot(a,a) theoretically returns sqrt(2)*abs(a), as shown in this example:

```
x = 1.271161e308;
```

```
y = x * sqrt(2)
y =
  1.7977e+308

y = hypot(x,x)
y =
  1.7977e+308
```

See Also

sqrt | abs | norm

Purpose

Imaginary unit

Syntax

```
i
a+bi
x+i*y
```

Description

As the basic imaginary unit `sqrt(-1)`, `i` is used to enter complex numbers. You can also use the character `j` as the imaginary unit. If you want to create a complex number without using `i` and `j`, you can use the `complex` function.

Since `i` is a function, it can be overridden and used as a variable. However, it is best to avoid using `i` and `j` for variable names if you intend to use them in complex arithmetic.

If desired, you can use the character `i` without a multiplication sign as a suffix in forming a complex numerical constant.

For speed and improved robustness, you can replace complex `i` and `j` by `1i`. For example, instead of using

```
a = i;
```

use

```
a = 1i;
```

Examples

```
Z = 2+3i
Z = x+i*y
Z = r*exp(i*theta)
```

See Also

`conj` | `imag` | `j` | `real`

Purpose

Incomplete Cholesky factorization

Syntax

```
L = ichol(A)
L = ichol(A,opts)
```

Description

`L = ichol(A)` performs the incomplete Cholesky factorization of `A` with zero-fill.

`L = ichol(A,opts)` performs the incomplete Cholesky factorization of `A` with options specified by `opts`.

By default, `ichol` references the lower triangle of `A` and produces lower triangular factors.

Input Arguments

A

Sparse matrix

opts

Structure with up to five fields:

Field Name	Summary	Description
`type`	Type of factorization	String indicating which flavor of incomplete Cholesky to perform. Valid values of this field are `'nofill'` and `'ict'`. The `'nofill'` variant performs *incomplete Cholesky with zero-fill* (`IC(0)`). The `'ict'` variant performs *incomplete Cholesky with threshold*

Field Name	Summary	Description
		dropping (ICT). The default value is 'nofill'.
droptol	Drop tolerance when type is 'ict'	Nonnegative scalar used as a drop tolerance when performing ICT. Elements which are smaller in magnitude than a local drop tolerance are dropped from the resulting factor except for the diagonal element which is never dropped. The local drop tolerance at step j of the factorization is norm(A(j:end,j),1)*droptol. 'droptol' is ignored if 'type' is 'nofill'. The default value is 0.
michol	Indicates whether to perform modified incomplete Cholesky	Indicates whether or not *modified incomplete Cholesky* (MIC) is performed. The field may be 'on' or 'off'. When performing MIC, the diagonal is compensated for dropped elements to enforce the relationship A*e = L*L'*e where e = ones(size(A,2),1)). The default value is 'off'.

Field Name	Summary	Description
diagcomp	Perform compensated incomplete Cholesky with the specified coefficient	Real nonnegative scalar used as a global diagonal shift alpha in forming the incomplete Cholesky factor. That is, instead of performing incomplete Cholesky on A, the factorization of A + alpha*diag(diag(A)) is formed. The default value is 0.
shape	Determines which triangle is referenced and returned	Valid values are 'upper' and 'lower'. If 'upper' is specified, only the upper triangle of A is referenced and R is constructed such that A is approximated by R'*R. If 'lower' is specified, only the lower triangle of A is referenced and L is constructed such that A is approximated by L*L'. The default value is 'lower'.

Tips

- The factor given by this routine may be useful as a preconditioner for a system of linear equations being solved by iterative methods such as pcg or minres.
- ichol works only for sparse square matrices

Examples

Incomplete Cholesky Factorization

This example generates an incomplete Cholesky factorization.

Start with a symmetric positive definite matrix, A:

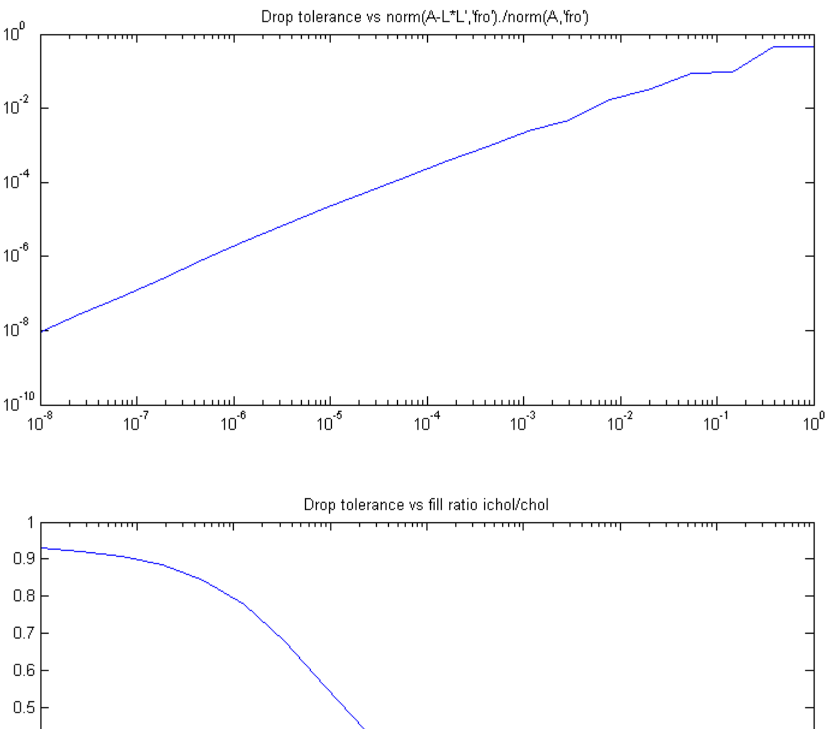
```
N = 100;
A = delsq(numgrid('S',N));
```

A is the two-dimensional, five-point discrete negative Laplacian on a 100-by-100 square grid with Dirichlet boundary conditions. The size of A is `98*98 = 9604` (not 10000 as the borders of the grid are used to impose the Dirichlet conditions).

The no-fill incomplete Cholesky factorization is a factorization which contains only nonzeros in the same position as A contains nonzeros. This factorization is extremely cheap to compute. Although the product `L*L'` is typically very different from A, the product `L*L'` will match A on its pattern up to round-off.

```
L = ichol(A);
norm(A-L*L','fro')./norm(A,'fro');
ans =
  9.1599e-002

norm(A-(L*L').*spones(A),'fro')./norm(A,'fro');
ans =
  4.9606e-017
```

`ichol` may also be used to generate incomplete Cholesky factorizations with threshold dropping. As the drop tolerance decreases, the factor tends to get more dense and the product `L*L'` tends to be a better approximation of A. The following plots show the relative error of the incomplete factorization plotted against the drop tolerance as well as the ratio of the density of the incomplete factors to the density of the complete Cholesky factor.



The relative error is typically on the same order as the drop tolerance, although this is not guaranteed.

Using a Preconditioner

Use the same matrix A as in the previous example:

```
N = 100;
A = delsq(numgrid('S',N));
```

Create an incomplete Cholesky factorization as a preconditioner for `pcg`. Use a constant vector as the right hand side. As a baseline, execute `pcg` without a preconditioner:

```
b = ones(size(A,1),1);
tol = 1e-6; maxit = 100;
[x0,fl0,rr0,it0,rv0] = pcg(A,b,tol,maxit);
```

Note that `fl0 = 1` indicating that `pcg` did not drive the relative residual to the requested tolerance in the maximum allowed iterations. Try the no-fill incomplete Cholesky factorization as a preconditioner:

```
L1 = ichol(A);
[x1,fl1,rr1,it1,rv1] = pcg(A,b,tol,maxit,L1,L1');
```

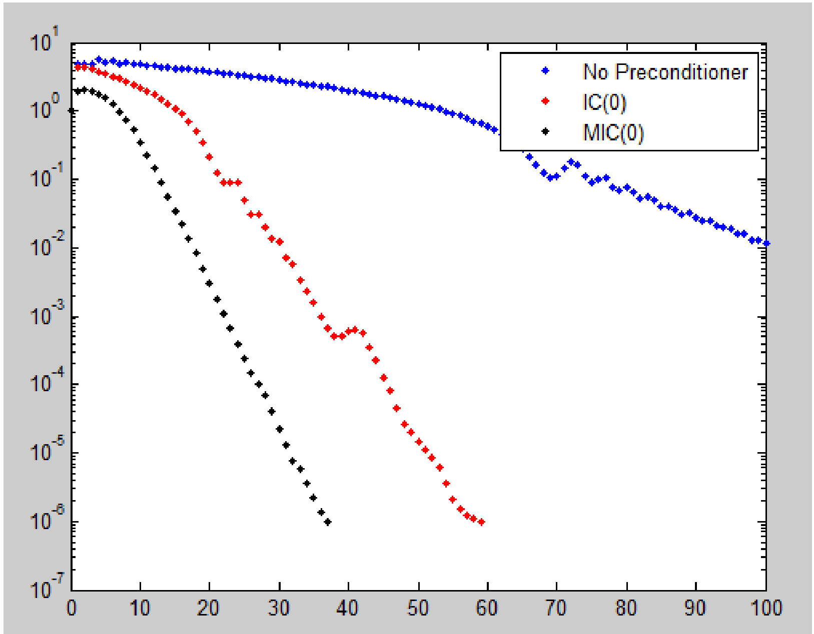
`fl1 = 0`, indicating that `pcg` converged to the requested tolerance and did so in 59 iterations (the value of `it1`). Since this matrix is a discretized Laplacian, however, using modified incomplete Cholesky can create a better preconditioner. A modified incomplete Cholesky factorization constructs an approximate factorization that preserves the action of the operator on the constant vector. That is, `norm(A*e – L*(L'*e))` will be approximately zero for `e = ones(size(A,2),1)` even though `norm(A-L*L','fro')/norm(A,'fro')` is not close to zero.

Note It is not necessary to specify `type` for this syntax since `nofill` is the default, but it is good practice.

```
opts.type = 'nofill'; opts.michol = 'on';
L2 = ichol(A,opts);
e = ones(size(A,2),1);
norm(A*e-L2*(L2'*e))
ans =
  3.7983e-014

[x2,fl2,rr2,it2,rv2] = pcg(A,b,tol,maxit,L2,L2');
```

pcg converges (`fl2 = 0`) but in only 38 iterations. Plotting all three convergence histories shows the convergence:

```
semilogy(0:maxit,rv0./norm(b),'b.');
hold on;
semilogy(0:it1,rv1./norm(b),'r.');
semilogy(0:it2,rv2./norm(b),'k.');
legend('No Preconditioner','IC(0)','MIC(0)');
```

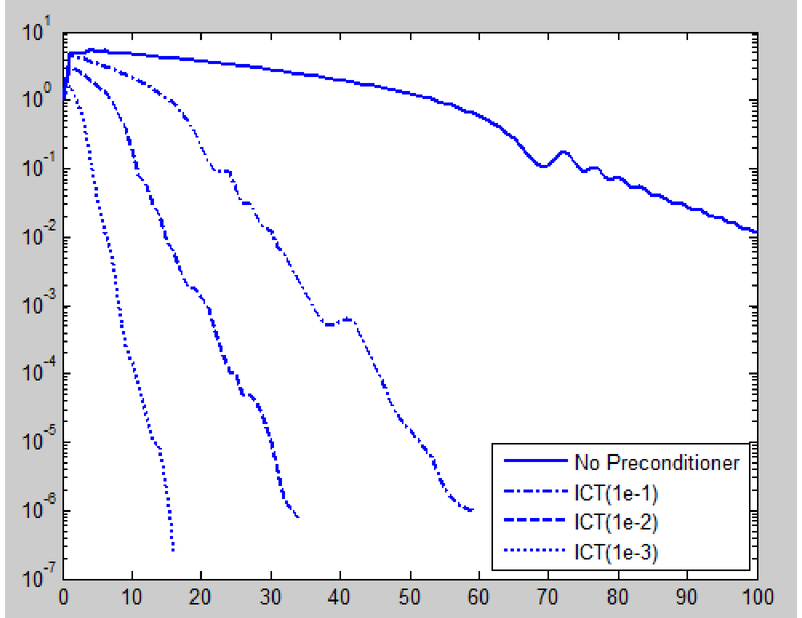


The plot shows that the modified incomplete Cholesky preconditioner creates a much faster convergence.

You can also try incomplete Cholesky factorizations with threshold dropping. The following plot shows convergence of `pcg` with preconditioners constructed with various drop tolerances.

```
L3 = ichol(A, struct('type','ict','droptol',1e-1));
[x3,fl3,rr3,it3,rv3] = pcg(A,b,tol,maxit,L3,L3');
```

```
L4 = ichol(A, struct('type','ict','droptol',1e-2));
[x4,fl4,rr4,it4,rv4] = pcg(A,b,tol,maxit,L4,L4');
L5 = ichol(A, struct('type','ict','droptol',1e-3));
[x5,fl5,rr5,it5,rv5] = pcg(A,b,tol,maxit,L5,L5');
figure; semilogy(0:maxit,rv0./norm(b),'b-','linewidth',2);
hold on;
semilogy(0:it3,rv3./norm(b),'b-.','linewidth',2);
semilogy(0:it4,rv4./norm(b),'b--','linewidth',2);
semilogy(0:it5,rv5./norm(b),'b:','linewidth',2);
legend('No Preconditioner','ICT(1e-1)','ICT(1e-2)', ...
   'ICT(1e-3)','Location','SouthEast');
```



Note the incomplete Cholesky preconditioner constructed with drop tolerance `1e-2` is denoted as `ICT(1e-2)`.

As with the zero-fill incomplete Cholesky, the threshold dropping factorization can benefit from modification (i.e. `opts.michol = 'on'`) since the matrix arises from an elliptic partial differential equation.

As with `MIC(0)`, the modified threshold based dropping incomplete Cholesky will preserve the action of the preconditioner on constant vectors, that is `norm(A*e-L*(L'*e))` will be approximately zero.

Using the diagcomp Option

This example illustrates using the `diagcomp` option of `ichol`. Incomplete Cholesky factorizations of positive definite matrices do not always exist. The following code constructs a random symmetric positive definite matrix and attempts to solve a linear system using `pcg`.

```
S = rng('default');
A = sprandsym(1000,1e-2,1e-4,1);
rng(S);
b = full(sum(A,2));
[x0,fl0,rr0,it0,rv0] = pcg(A,b,1e-6,100);
```

Since convergence is not attained, try to construct an incomplete Cholesky preconditioner:

```
L = ichol(A,struct('type','ict','droptol',1e-3));
Error using ichol
Encountered nonpositive pivot.
```

If `ichol` breaks down as above, you can use the `diagcomp` option to construct a shifted incomplete Cholesky factorization. That is, instead of constructing `L` such that `L*L'` approximates `A`, `ichol` with diagonal compensation constructs `L` such that `L*L'` approximates `M = A + alpha*diag(diag(A))` without explicitly forming `M`. As incomplete factorizations always exist for diagonally dominant matrices, `alpha` can be found to make `M` diagonally dominant:

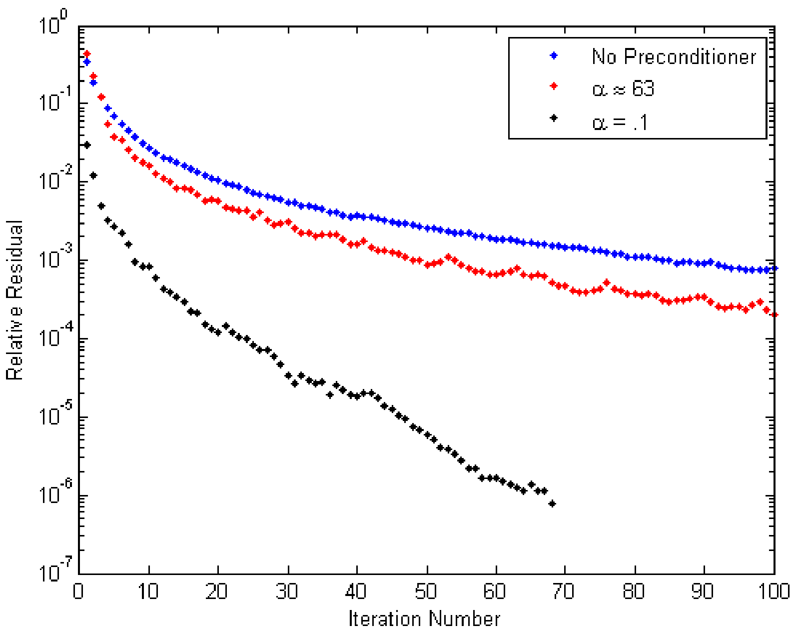
```
alpha = max(sum(abs(A),2)./diag(A))-2
alpha =
   62.9341

L1 = ichol(A, struct('type','ict','droptol',1e-3,'diagcomp',alpha));
[x1,fl1,rr1,it1,rv1] = pcg(A,b,1e-6,100,L1,L1');
```

Here, pcg still fails to converge to the desired tolerance within the desired number of iterations, but as the plot below shows, convergence is better for pcg with this preconditioner than with no preconditioner. Choosing a smaller alpha may help. With some experimentation, we can settle on an appropriate value for alpha:

```
alpha = .1;
L2 = ichol(A, struct('type','ict','droptol',1e-3,'diagcomp',alpha));
[x2,fl2,rr2,it2,rv2] = pcg(A,b,1e-6,100,L2,L2');
```

Now, pcg converges and a plot can show the convergence histories with each preconditioner:

```
semilogy(0:100,rv0./norm(b),'b.');
hold on;
semilogy(0:100,rv1./norm(b),'r.');
semilogy(0:it2,rv2./norm(b),'k.');
legend('No Preconditioner','\alpha \approx 63','\alpha = .1');
xlabel('Iteration Number');
ylabel('Relative Residual');
```



References

[1] Saad, Yousef. “Preconditioning Techniques.” *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, 1996.

[2] Manteuffel, T.A. “An incomplete factorization technique for positive definite linear systems.” *Math. Comput.* 34, 473–497, 1980.

See Also

`ilu` | `chol` | `pcg` | `minres`

Purpose

Integer division with rounding option

Syntax

```
C = idivide(A, B, opt)
C = idivide(A, B)
C = idivide(A, B, 'fix')
C = idivide(A, B, 'round')
C = idivide(A, B, 'floor')
C = idivide(A, B, 'ceil')
```

Description

`C = idivide(A, B, opt)` is the same as `A./B` for integer classes except that fractional quotients are rounded to integers using the optional rounding mode specified by `opt`. The default rounding mode is `'fix'`. Inputs `A` and `B` must be real and must have the same dimensions unless one is a scalar. At least one of the arguments `A` and `B` must belong to an integer class, and the other must belong to the same integer class or be a scalar double. The result `C` belongs to the integer class.

`C = idivide(A, B)` is the same as `A./B` except that fractional quotients are rounded toward zero to the nearest integers.

`C = idivide(A, B, 'fix')` is the same as the syntax shown immediately above.

`C = idivide(A, B, 'round')` is the same as `A./B` for integer classes. Fractional quotients are rounded to the nearest integers.

`C = idivide(A, B, 'floor')` is the same as `A./B` except that fractional quotients are rounded toward negative infinity to the nearest integers.

`C = idivide(A, B, 'ceil')` is the same as `A./B` except that the fractional quotients are rounded toward infinity to the nearest integers.

Examples

```
a = int32([-2 2]);
b = int32(3);

idivide(a,b)            % Returns [0 0]
idivide(a,b,'floor')    % Returns [-1 0]
idivide(a,b,'ceil')     % Returns [0 1]
```

```
idivide(a,b,'round')     % Returns [-1 1]
```

See Also

ldivide | rdivide | ceil | floor | fix | round

Purpose

Execute statements if condition is true

Syntax

```
if expression
   statements
elseif expression
   statements
else
   statements
end
```

Description

`if expression, statements, end` evaluates an expression, and executes a group of statements when the expression is true.

`elseif` and `else` are optional, and execute statements only when previous expressions in the `if` block are false. An `if` block can include multiple `elseif` statements.

An evaluated expression is true when the result is nonempty and contains all nonzero elements (logical or real numeric). Otherwise, the expression is false.

Expressions can include relational operators (such as < or ==) and logical operators (such as `&&`, `||`, or ~). MATLAB evaluates compound expressions from left to right, adhering to operator precedence rules.

Tips

- You can nest any number of `if` statements. Each `if` statement requires an `end` keyword.
- Avoid adding a space within the `elseif` keyword (`else if`). The space creates a nested `if` statement that requires its own `end` keyword.
- Within an `if` or `while` expression, all logical operators, including `|` and `&`, short-circuit. That is, if the first part of the expression determines a true or false result, MATLAB does not evaluate the second part of the expression.

Examples

Assign to a matrix values that depend on their indices:

```
% Preallocate a matrix
nrows = 10;
ncols = 10;
myData = ones(nrows, ncols);

% Loop through the matrix
for r = 1:nrows
   for c = 1:ncols

      if r == c
         myData(r,c) = 2;
      elseif abs(r - c) == 1
         myData(r,c) = -1;
      else
         myData(r,c) = 0;
      end

   end
end
```

Respond to command-line input. Because the input string could be more than one character, use `strcmp` rather than == to test for equality:

```
reply = input('Would you like to see an echo? (y/n): ', 's');
if strcmp(reply,'y')
  disp(reply)
end
```

Find the indices of values in a vector that are greater than a specified limit:

```
A = rand(1,10);
limit = .75;

B = (A > limit);   % B is a vector of logical values
```

```
if any(B)
  fprintf('Indices of values > %4.2f: \n', limit);
  disp(find(B))
else
  disp('All values are below the limit.')
end
```

Concatenate two variables when they are the same size. To avoid an error when the variables have different dimensions, compare the sizes using `isequal` rather than the == operator:

```
A = ones(2,3);            % Two-dimensional array
B = rand(3,4,5);          % Three-dimensional array

if isequal(size(A), size(B))
   C = [A; B];
else
   warning('A and B are not the same size.');
   C = [];
end
```

Take advantage of short-circuiting to avoid error or warning messages:

```
x = 42;
if exist('myfunction.m') && (myfunction(x) >= pi)
   disp('Condition is true')
end
```

See Also

`for` | `while` | `switch` | `return`

Tutorials

- “Relational Operators”
- “Logical Operators”

Purpose

Inverse fast Fourier transform

Syntax

```
y = ifft(X)
y = ifft(X,n)
y = ifft(X,[],dim)
y = ifft(X,n,dim)
y = ifft(..., 'symmetric')
y = ifft(..., 'nonsymmetric')
```

Description

`y = ifft(X)` returns the inverse discrete Fourier transform (DFT) of vector `X`, computed with a fast Fourier transform (FFT) algorithm. If `X` is a matrix, `ifft` returns the inverse DFT of each column of the matrix.

`ifft` tests `X` to see whether vectors in `X` along the active dimension are *conjugate symmetric*. If so, the computation is faster and the output is real. An `N`-element vector `x` is conjugate symmetric if `x(i) = conj(x(mod(N-i+1,N)+1))` for each element of `x`.

If `X` is a multidimensional array, `ifft` operates on the first non-singleton dimension.

`y = ifft(X,n)` returns the `n`-point inverse DFT of vector `X`.

`y = ifft(X,[],dim)` and `y = ifft(X,n,dim)` return the inverse DFT of `X` across the dimension `dim`.

`y = ifft(..., 'symmetric')` causes `ifft` to treat `X` as conjugate symmetric along the active dimension. This option is useful when `X` is not exactly conjugate symmetric, merely because of round-off error.

`y = ifft(..., 'nonsymmetric')` is the same as calling `ifft(...)` without the argument `'nonsymmetric'`.

For any `X`, `ifft(fft(X))` equals `X` to within roundoff error.

Algorithms

The algorithm for `ifft(X)` is the same as the algorithm for `fft(X)`, except for a sign change and a scale factor of `n = length(X)`. As for `fft`, the execution time for `ifft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have

only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `ifft` using the utility function `fftw`, which controls how MATLAB software optimizes the algorithm used to compute an FFT of a particular size and dimension.

Data Type Support

`ifft` supports inputs of data types `double` and `single`. If you call `ifft` with the syntax `y = ifft(X, ...)`, the output `y` has the same data type as the input `X`.

See Also

`fft` | `fft2` | `ifft2` | `ifftn` | `ifftshift` | `fftw` | `ifft2` | `ifftn` | `dftmtx` | `freqz`

Purpose

2-D inverse fast Fourier transform

Syntax

```
Y = ifft2(X)
Y = ifft2(X,m,n)
y = ifft2(..., 'symmetric')
y = ifft2(..., 'nonsymmetric')
```

Description

`Y = ifft2(X)` returns the two-dimensional inverse discrete Fourier transform (DFT) of `X`, computed with a fast Fourier transform (FFT) algorithm. The result `Y` is the same size as `X`.

`ifft2` tests `X` to see whether it is *conjugate symmetric*. If so, the computation is faster and the output is real. An `M`-by-`N` matrix `X` is conjugate symmetric if `X(i,j) = conj(X(mod(M-i+1, M) + 1, mod(N-j+1, N) + 1))` for each element of `X`.

`Y = ifft2(X,m,n)` returns the `m`-by-`n` inverse fast Fourier transform of matrix `X`.

`y = ifft2(..., 'symmetric')` causes `ifft2` to treat `X` as conjugate symmetric. This option is useful when `X` is not exactly conjugate symmetric, merely because of round-off error.

`y = ifft2(..., 'nonsymmetric')` is the same as calling `ifft2(...)` without the argument `'nonsymmetric'`.

For any `X`, `ifft2(fft2(X))` equals `X` to within roundoff error.

Algorithms

The algorithm for `ifft2(X)` is the same as the algorithm for `fft2(X)`, except for a sign change and scale factors of `[m,n] = size(X)`. The execution time for `ifft2` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `ifft2` using the utility function `fftw`, which controls how MATLAB software optimizes the algorithm used to compute an FFT of a particular size and dimension.

Data Type Support

`ifft2` supports inputs of data types `double` and `single`. If you call `ifft2` with the syntax `y = ifft2(X, ...)`, the output `y` has the same data type as the input `X`.

See Also

`dftmtx` | `freqz` | `fft2` | `fftw` | `fftshift` | `ifft` | `ifftn` | `ifftshift`

ifftn

Purpose

N-D inverse fast Fourier transform

Syntax

```
Y = ifftn(X)
Y = ifftn(X,siz)
y = ifftn(..., 'symmetric')
y = ifftn(..., 'nonsymmetric')
```

Description

`Y = ifftn(X)` returns the n-dimensional inverse discrete Fourier transform (DFT) of `X`, computed with a multidimensional fast Fourier transform (FFT) algorithm. The result `Y` is the same size as `X`.

`ifftn` tests `X` to see whether it is *conjugate symmetric*. If so, the computation is faster and the output is real. An `N1`-by-`N2`-by- ... `Nk` array `X` is conjugate symmetric if

```
X(i1,i2, ...,ik) = conj(X(mod(N1-i1+1,N1)+1, mod(N2-i2+1,N2)+1,
... mod(Nk-ik+1,Nk)+1))
```

for each element of `X`.

`Y = ifftn(X,siz)` pads `X` with zeros, or truncates `X`, to create a multidimensional array of size `siz` before performing the inverse transform. The size of the result `Y` is `siz`.

`y = ifftn(..., 'symmetric')` causes `ifftn` to treat `X` as conjugate symmetric. This option is useful when `X` is not exactly conjugate symmetric, merely because of round-off error.

`y = ifftn(..., 'nonsymmetric')` is the same as calling `ifftn(...)` without the argument `'nonsymmetric'`.

Tips

For any `X`, `ifftn(fftn(X))` equals `X` within roundoff error.

Algorithms

`ifftn(X)` is equivalent to

```
Y = X;
for p = 1:length(size(X))
    Y = ifft(Y,[],p);
end
```

This computes in-place the one-dimensional inverse DFT along each dimension of `X`.

The execution time for `ifftn` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

Note You might be able to increase the speed of `ifftn` using the utility function `fftw`, which controls how MATLAB software optimizes the algorithm used to compute an FFT of a particular size and dimension.

Data Type Support

`ifftn` supports inputs of data types `double` and `single`. If you call `ifftn` with the syntax `y = ifftn(X, ...)`, the output `y` has the same data type as the input `X`.

See Also

`fftn` | `fftw` | `ifft` | `ifft2` | `ifftshift`

ifftshift

Purpose

Inverse FFT shift

Syntax

```
ifftshift(X)
ifftshift(X,dim)
```

Description

`ifftshift(X)` swaps the left and right halves of the vector `X`. For matrices, `ifftshift(X)` swaps the first quadrant with the third and the second quadrant with the fourth. If `X` is a multidimensional array, `ifftshift(X)` swaps "half-spaces" of `X` along each dimension.

`ifftshift(X,dim)` applies the `ifftshift` operation along the dimension `dim`.

Note `ifftshift` undoes the results of `fftshift`. If the matrix `X` contains an odd number of elements, `ifftshift(fftshift(X))` must be done to obtain the original `X`. Simply performing `fftshift(X)` twice will not produce `X`.

See Also

`fft` | `fft2` | `fftn` | `fftshift`

Purpose

Sparse incomplete LU factorization

Syntax

```
ilu(A,setup)
[L,U] = ilu(A,setup)
[L,U,P] = ilu(A,setup)
```

Description

`ilu` produces a unit lower triangular matrix, an upper triangular matrix, and a permutation matrix.

`ilu(A,setup)` computes the incomplete LU factorization of `A`. `setup` is an input structure with up to five setup options. The fields must be named exactly as shown in the table below. You can include any number of these fields in the structure and define them in any order. Any additional fields are ignored.

Field Name	Description
`type`	Type of factorization. Values for `type` include: • `'nofill'`—Performs ILU factorization with `0` level of fill in, known as ILU(0). With `type` set to `'nofill'`, only the `milu` setup option is used; all other fields are ignored. • `'crout'`—Performs the Crout version of ILU factorization, known as ILUC. With `type` set to `'crout'`, only the `droptol` and `milu` setup options are used; all other fields are ignored. • `'ilutp'` (default)—Performs ILU factorization with threshold and pivoting. If `type` is not specified, the ILU factorization with pivoting ILUTP is performed. Pivoting is never performed with `type` set to `'nofill'` or `'crout'`.
`droptol`	Drop tolerance of the incomplete LU factorization. `droptol` is a non-negative scalar. The default value is `0`, which produces the complete LU factorization.

Field Name	Description

	The nonzero entries of `U` satisfy
	
	`abs(U(i,j)) >= droptol*norm((A:,j)),`
	
	with the exception of the diagonal entries, which are retained regardless of satisfying the criterion. The entries of `L` are tested against the local drop tolerance before being scaled by the pivot, so for nonzeros in `L`
	
	`abs(L(i,j)) >= droptol*norm(A(:,j))/U(j,j).`

`milu`	Modified incomplete LU factorization. Values for `milu` include: • `'row'`—Produces the row-sum modified incomplete LU factorization. Entries from the newly-formed column of the factors are subtracted from the diagonal of the upper triangular factor, `U`, preserving column sums. That is, `A*e = L*U*e`, where `e` is the vector of ones. • `'col'`—Produces the column-sum modified incomplete LU factorization. Entries from the newly-formed column of the factors are subtracted from the diagonal of the upper triangular factor, `U`, preserving column sums. That is, `e'*A = e'*L*U`. • `'off'` (default)—No modified incomplete LU factorization is produced.
`udiag`	If `udiag` is `1`, any zeros on the diagonal of the upper triangular factor are replaced by the local drop tolerance. The default is `0`.
`thresh`	Pivot threshold between `0` (forces diagonal pivoting) and `1`, the default, which always chooses the maximum magnitude entry in the column to be the pivot.

`ilu(A,setup)` returns `L+U-speye(size(A))`, where `L` is a unit lower triangular matrix and `U` is an upper triangular matrix.

`[L,U] = ilu(A,setup)` returns a unit lower triangular matrix in `L` and an upper triangular matrix in `U`.

`[L,U,P] = ilu(A,setup)` returns a unit lower triangular matrix in `L`, an upper triangular matrix in `U`, and a permutation matrix in `P`.

Tips

These incomplete factorizations may be useful as preconditioners for a system of linear equations being solved by iterative methods such as BICG (BiConjugate Gradients), GMRES (Generalized Minimum Residual Method).

Limitations

`ilu` works on sparse square matrices only.

Examples

Start with a sparse matrix and compute the LU factorization.

```
A = gallery('neumann', 1600) + speye(1600);
setup.type = 'crout';
setup.milu = 'row';
setup.droptol = 0.1;
[L,U] = ilu(A,setup);
e = ones(size(A,2),1);
norm(A*e-L*U*e)

ans =

  1.4251e-014
```

This shows that `A` and `L*U`, where `L` and `U` are given by the modified Crout `ILU`, have the same row-sum.

Start with a sparse matrix and compute the LU factorization.

```
A = gallery('neumann', 1600) + speye(1600);
setup.type = 'nofill';
nnz(A)
```

```
ans =

        7840

nnz(lu(A))
ans =

      126478

nnz(ilu(A,setup))
ans =

        7840
```

This shows that `A` has `7840` nonzeros, the complete LU factorization has `126478` nonzeros, and the incomplete LU factorization, with `0` level of fill-in, has `7840` nonzeros, the same amount as `A`.

References

[1] Saad, Yousef, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996, Chapter 10 - Preconditioning Techniques.

See Also

`bicg` | `cholinc` | `gmres` | `luinc`

Purpose

Convert image to movie frame

Syntax

```
f = im2frame(X,map)
f = im2frame(X)
```

Description

`f = im2frame(X,map)` converts the indexed image `X` and associated colormap `map` into a movie frame `f`. If `X` is a truecolor (m-by-n-by-3) image, then `map` is optional and has no effect.

Typical usage:

```
M(1) = im2frame(X1,map);
M(2) = im2frame(X2,map);
 ...
M(n) = im2frame(Xn,map);
movie(M)
```

`f = im2frame(X)` converts the indexed image `X` into a movie frame `f` using the current colormap if `X` contains an indexed image.

See Also

`frame2im` | `movie`

im2java

Purpose

Convert image to Java image

Syntax

```
jimage = im2java(I)
jimage = im2java(X,MAP)
jimage = im2java(RGB)
```

Description

To work with a MATLAB image in the Java environment, you must convert the image from its MATLAB representation into an instance of the Java image class, `java.awt.Image`.

`jimage = im2java(I)` converts the intensity image `I` to an instance of the Java image class, `java.awt.Image`.

`jimage = im2java(X,MAP)` converts the indexed image `X`, with colormap `MAP`, to an instance of the Java image class, `java.awt.Image`.

`jimage = im2java(RGB)` converts the RGB image `RGB` to an instance of the Java image class, `java.awt.Image`.

Class Support

The input image can be of class `uint8`, `uint16`, or `double`.

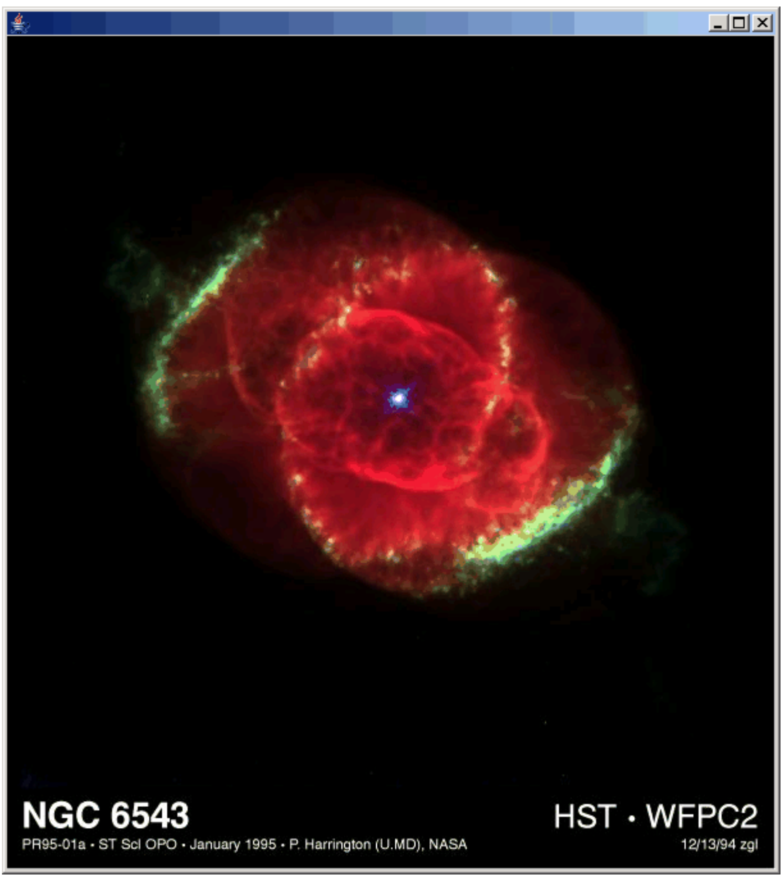
Note Java requires `uint8` data to create an instance of the Java image class, `java.awt.Image`. If the input image is of class `uint8`, `jimage` contains the same `uint8` data. If the input image is of class `double` or `uint16`, `im2java` makes an equivalent image of class `uint8`, rescaling or offsetting the data as necessary, and then converts this `uint8` representation to an instance of the Java image class, `java.awt.Image`.

Examples

This example reads an image into the MATLAB workspace and then uses `im2java` to convert it into an instance of the Java image class.

```
I = imread('ngc6543a.jpg');
javaImage = im2java(I);
frame = javax.swing.JFrame;
icon = javax.swing.ImageIcon(javaImage);
label = javax.swing.JLabel(icon);
```

```
frame.getContentPane.add(label);
frame.pack
frame.show
```



imag

Purpose

Imaginary part of complex number

Syntax

```
Y = imag(Z)
```

Description

`Y = imag(Z)` returns the imaginary part of the elements of array `Z`.

Examples

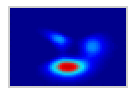
```
imag(2+3i)

ans =

    3
```

See Also

`conj` | `i` | `j` | `real`

Purpose

Display image object



Syntax

```
image(C)
image(x,y,C)
image(x,y,C,'PropertyName',PropertyValue,...)
image('PropertyName',PropertyValue,...)
handle = image(...)
```

Properties

For a list of properties, see Image Properties.

Description

`image` creates an image graphics object by interpreting each element in a matrix as an index into the figure's colormap or directly as RGB values, depending on the data specified.

The `image` function has two forms:

- A high-level function that calls `newplot` to determine where to draw the graphics objects and sets the following axes properties:
  - `XLim` and `YLim` to enclose the image
  - `Layer` to `top` to place the image in front of the tick marks and grid lines
  - `YDir` to `reverse`
  - `View` to `[0 90]`
- A low-level function that adds the image to the current axes without calling `newplot`. The low-level function argument list can contain only property name/property value pairs.

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see set and get for examples of how to specify these data types).

`image(C)` displays matrix `C` as an image. Each element of `C` specifies the color of a rectangular segment in the image.

`image(x,y,C)`, where `x` and `y` are two-element vectors, specifies the range of the *x*- and *y*-axis labels, but produces the same image as `image(C)`. This can be useful, for example, if you want the axis tick labels to correspond to real physical dimensions represented by the image. If `x(1) > x(2)` or `y(1) > y(2)`, the image is flipped left-right or up-down, respectively. It can also be useful when you want to place the image within a set of axes already created. In this case, use `hold on` with the current figure and enter *x* and *y* values corresponding to the corners of the desired image location. The image is stretched and oriented as applicable.

`image(x,y,C,'PropertyName',PropertyValue,...)` is a high-level function that also specifies property name/property value pairs. For a description of the properties, see Image Properties. This syntax calls `newplot` before drawing the image.

`image('PropertyName',PropertyValue,...)` is the low-level syntax of the `image` function. It specifies only property name/property value pairs as input arguments.

`handle = image(...)` returns the handle of the image object it creates. You can obtain the handle with all forms of the `image` function.

Tips

Image data can be either indexed or true color. An indexed image stores colors as an array of indices into the figure colormap. A true color image does not use a colormap; instead, the color values for each pixel are stored directly as RGB triplets. In MATLAB graphics, the `CData` property of a truecolor image object is a three-dimensional (`m`-by-`n`-by-3) array. This array consists of three `m`-by-`n` matrices (representing the red, green, and blue color planes) concatenated along the third dimension.

The `imread` function reads image data into MATLAB arrays from graphics files in various standard formats, such as TIFF. You can write MATLAB image data to graphics files using the `imwrite` function. `imread` and `imwrite` both support a variety of graphics file formats and compression schemes.

When you read image data into the MATLAB workspace using `imread`, the data is usually stored as an array of 8-bit integers. However, `imread` also supports reading 16-bit-per-pixel data from TIFF and PNG files. These are more efficient storage methods than the double-precision (64-bit) floating-point numbers that MATLAB typically uses. However, it is necessary to interpret 8-bit and 16-bit image data differently from 64-bit data. This table summarizes these differences.

You cannot interactively pan or zoom outside the *x*-limits or *y*-limits of an image, unless the axes limits are already been set outside the bounds of the image, in which case there is no such restriction. If other objects (such as lineseries) occupy the axes and extend beyond the bounds of the image, you can pan or zoom to the bounds of the other objects, but no further.

Image Type	Double-Precision Data (double Array)	8-Bit Data (uint8 Array) 16-Bit Data (uint16 Array)
Indexed (colormap)	Image is stored as a two-dimensional (`m`-by-`n`) array of integers in the range [1, `length(colormap)`]; `colormap` is an `m`-by-3 array of floating-point values in the range [0, 1].	Image is stored as a two-dimensional (`m`-by-`n`) array of integers in the range [0, 255] (`uint8`) or [0, 65535] (`uint16`); `colormap` is an `m`-by-3 array of floating-point values in the range [0, 1].
True color (RGB)	Image is stored as a three-dimensional (`m`-by-`n`-by-3) array of floating-point values in the range [0, 1].	Image is stored as a three-dimensional (`m`-by-`n`-by-3) array of integers in the range [0, 255] (`uint8`) or [0, 65535] (`uint16`).

By default, `image` plots the *y*-axis from lowest to highest value, top to bottom. To reverse this, type `set(gca,'YDir','normal')`. This will reverse both the *y*-axis and the image.

Indexed Images

In an indexed image of class `double`, the value 1 points to the first row in the colormap, the value 2 points to the second row, and so on. In a `uint8` or `uint16` indexed image, there is an offset; the value 0 points to the first row in the colormap, the value 1 points to the second row, and so on.

If you want to convert a `uint8` or `uint16` indexed image to `double`, you need to add 1 to the result. For example,

```
X64 = double(X8) + 1;
```

or

```
X64 = double(X16) + 1;
```

To convert from `double` to `uint8` or `uint16`, you need to first subtract 1, and then use `round` to ensure all the values are integers.

```
X8 = uint8(round(X64 - 1));
```

or

```
X16 = uint16(round(X64 - 1));
```

When you write an indexed image using `imwrite`, values are automatically converted if necessary.

Colormaps

MATLAB colormaps are always `m`-by-3 arrays of double-precision floating-point numbers in the range [0, 1]. In most graphics file formats, colormaps are stored as integers, but MATLAB colormaps cannot have integer values. `imread` and `imwrite` automatically convert colormap values when reading and writing files.

True Color Images

In a true color image of class `double`, the data values are floating-point numbers in the range [0, 1]. In a true color image of class `uint8`, the

data values are integers in the range [0, 255], and for true color images of class `uint16` the data values are integers in the range [0, 65535].

If you want to convert a true color image from one data type to the other, you must rescale the data. For example, this statement converts a `uint8` true color image to `double`.

```
RGB64 = double(RGB8)/255;
```

or for `uint16` images,

```
RGB64 = double(RGB16)/65535;
```

This statement converts a `double` true color image to `uint8`:

```
RGB8 = uint8(round(RGB64*255));
```

or to obtain `uint16` images, type

```
RGB16 = uint16(round(RGB64*65535));
```

When you write a true color image using `imwrite`, values are automatically converted if necessary.

Examples

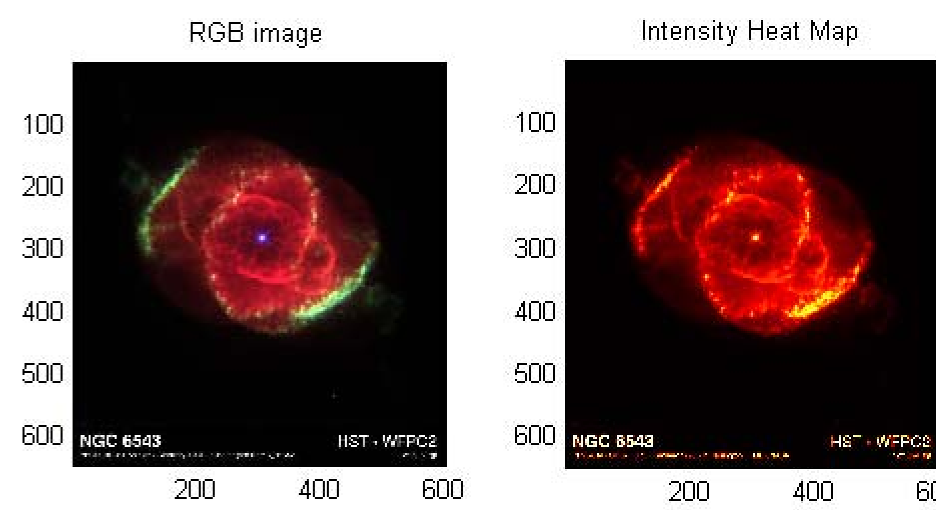
Load a mat-file containing a photograph of a colorful primate. Display the indexed image using its associated colormap.

```
load mandrill
figure('color','k')
image(X)
colormap(map)
axis off          % Remove axis ticks and numbers
axis image        % Set aspect ratio to obtain
square pixels
```

Load a JPEG image file of the Cat's Eye Nebula from the Hubble Space Telescope (image courtesy NASA). Display the original image using its RGB color values (left) as a subplot. Create a linked subplot (same size and scale) to display the transformed intensity image as a heat map (right).

```
figure
ax(1) = subplot(1,2,1);
rgb = imread('ngc6543a.jpg');
```

```
image(rgb); title('RGB image')
ax(2) = subplot(1,2,2);
im = mean(rgb,3);
image(im); title('Intensity Heat Map')
colormap(hot(256))
linkaxes(ax,'xy')
axis(ax,'image')
```



Setting Default Properties

You can set default image properties on the axes, figure, and `root object` levels:

```
set(0,'DefaultImageProperty',PropertyValue...)
set(gcf,'DefaultImageProperty',PropertyValue...)
set(gca,'DefaultImageProperty',PropertyValue...)
```

where *`Property`* is the name of the image property and `PropertyValue` is the value you are specifying. Use `set` and `get` to access image properties.

Tutorials

For more information, see "Working with Images in MATLAB Graphics" .

See Also

`colormap` | `imagesc` | `imfinfo` | `imread` | `imwrite` | `newplot` | `pcolor` | `surface` | Image Properties

Purpose

Define image properties

Creating Image Objects

Use `image` to create image objects.

Modifying Properties

You can set and query graphics object properties in two ways:

- "The Property Editor" is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see "Setting Default Property Values".

See "Core Graphics Objects" for general information about this type of object.

Image Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`AlphaData`

`m-by-n` matrix of `double` or `uint8`

Transparency data. A matrix of non-NaN values specifying the transparency of each face or vertex of the object. The `AlphaData` can be of class `double` or `uint8`.

MATLAB software determines the transparency in one of the following ways:

- Using the elements of `AlphaData` as transparency values (`AlphaDataMapping` set to `none`)
- Using the elements of `AlphaData` as indices into the current alphamap (`AlphaDataMapping` set to `direct`)

- Scaling the elements of `AlphaData` to range between the minimum and maximum values of the axes `ALim` property (`AlphaDataMapping` set to `scaled`, the default)

`AlphaDataMapping`
`none | direct | {scaled}`

Transparency mapping method. Determines how MATLAB interprets indexed alpha data. Values for this property are:

- `none` — The transparency values of `AlphaData` are between `0` and `1` or are clamped to this range.
- `scaled` — Transform the `AlphaData` to span the portion of the alphamap indicated by the axes `ALim` property, linearly mapping data values to alpha values (the default).
- `direct` — Use the `AlphaData` as indices directly into the alphamap. When not scaled, the data are usually integer values ranging from `1` to `length(alphamap)`. MATLAB maps values less than `1` to the first alpha value in the alphamap, and values greater than `length(alphamap)` to the last alpha value in the alphamap. Values with a decimal portion are fixed to the nearest, lower integer. If `AlphaData` is an array of `uint8` integers, then the indexing begins at `0` (that is, MATLAB maps a value of `0` to the first alpha value in the alphamap).

`Annotation`
`hg.Annotation` object (read-only)

Handle of `Annotation` object. The `Annotation` property enables you to specify whether this image object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the image object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Represent this image object in a legend (default)
`off`	Do not include this image object in a legend
`children`	Same as `on` because image objects do not have children

Setting the IconDisplayStyle property

Set the `IconDisplayStyle` of a graphics object with handle `hobj` to `off`:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Using the IconDisplayStyle property

See "Controlling Legends" for more information and examples.

`BeingDeleted`
`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to

perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`
: `cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`
: string | function handle

Button press callback function. Executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be

- A string that is a valid MATLAB expression
- The name of a MATLAB file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`CData`

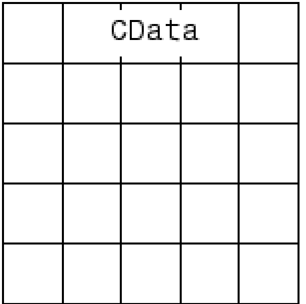
matrix | `m`-by-`n`-by-`3` array

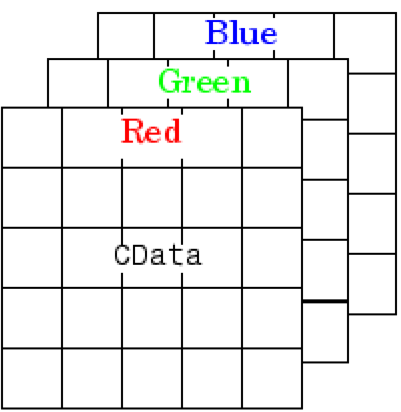
The image data. A matrix or 3-D array of values specifying the color of each rectangular area defining the image. `image(C)` assigns the values of `C` to `CData`. MATLAB determines the coloring of the image in one of three ways:

- Using the elements of `CData` as indices into the current colormap (the default) (`CDataMapping` set to `direct`)
- Scaling the elements of `CData` to range between the values `min(get(gca,'CLim'))` and `max(get(gca,'CLim'))` (`CDataMapping` set to `scaled`)
- Interpreting the elements of `CData` directly as RGB values (true color specification)

Note that the behavior of NaNs in image `CData` is not defined. See the image `AlphaData` property for information on using transparency with images.

A true color specification for `CData` requires an `m`-by-`n`-by-`3` array of RGB values. The first page contains the red component, the second page the green component, and the third page the blue component of each element in the image. RGB values range from `0` to `1`. The following picture illustrates the relative dimensions of `CData` for the two color models.

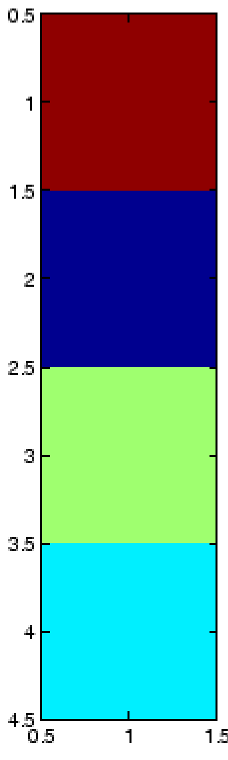




If CData has only one row or column, the height or width respectively is always one data unit and is centered about the first YData or XData element respectively. For example, using a 4-by-1 matrix of random data:

```
C = rand(4,1);
image(C,'CDataMapping','scaled')
axis image
```

produces



`CDataMapping`
`scaled | {direct}`

Direct or scaled indexed colors. Determines whether MATLAB interprets the values in `CData` as indices into the figure colormap (the default) or scales the values according to the values of the axes `CLim` property.

When `CDataMapping` is `direct`, the values of `CData` should be in the range 1 to `length(get(gcf,'Colormap'))`. If you use true color specification for `CData`, this property has no effect. If `CData` is of type `logical`, values of `0` index the first color of the colormap and values of `1` index the second color.

`Children`
handle

The empty matrix; image objects have no children.

`Clipping`
`{on} | off`

Clipping mode. By default, MATLAB clips images to the axes rectangle. If you set `Clipping` to `off`, the image can be displayed outside the axes rectangle. For example, if you create an image, set `hold` to `on`, freeze axis scaling (with `axis manual`), and then create a larger image, it extends beyond the axis limits.

`CreateFcn`
string | function handle

Callback routine executed during object creation. This property defines a callback routine that executes when MATLAB creates an image object. You must define this property as a default value for images or in a call to the `image` function to create a new image object. For example, the statement:

```
set(0,'DefaultImageCreateFcn','axis image')
```

defines a default value on the root level that sets the aspect ratio and the axis limits so the image has square pixels. MATLAB executes this routine after setting all image properties. Setting this property on an existing image object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See “Function Handle Callbacks” for information on how to use function handles to define the callback function.

`DeleteFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback executed during object deletion. Executes when this object is deleted (for example, this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values. The default is an empty array.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`

string

String used by legend. The `legend` function uses the `DisplayName` property to label the image object in the legend. The default is an empty string.

- If you specify string arguments with the `legend` function, MATLAB set `DisplayName` to the corresponding string and uses that string for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, MATLAB set `DisplayName` to the edited string.

- If you specify a string for the DisplayName property and create the legend using the figure toolbar, then MATLAB uses the string defined by DisplayName.
- To add a legend programmatically that uses the DisplayName string, call legend with the toggle or show option.

See "Controlling Legends" for more information and examples.

EraseMode
`{normal} | none | xor | background`

Erase mode. Controls the technique MATLAB uses to draw and erase objects. Use alternative erase modes for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- normal — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- none — Do not erase the object when it is moved or destroyed. While the object is still visible on the screen after erasing with EraseMode none, you cannot print it because MATLAB stores no information about its former location.
- xor — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object's color depends on the color of whatever is beneath it on the display.
- background — Erase the object by redrawing it in the axes background Color, or the figure background Color if the axes Color property is none. This damages objects that are behind the erased object, but properly colors the erased object.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors (for example, performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible.
- `callback` — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties. See also `findall`.

Handle Validity

Hidden handles are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`
`{on} | off`

Selectable by mouse click. Determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the objects that compose the area graph. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the `Interruptible` property affects only the callbacks for the`ButtonDownFcn` property. A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both the callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.

- If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see “Control Callback Execution and Interruption”.

`Parent`

handle of parent axes, hggroup, or hgtransform

Parent of object. Handle of the object’s parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See “Objects That Can Contain Other Objects” for more information on parenting graphics objects.

`Selected`

`on | {off}`

Object selection state. When you set this property to `on`, MATLAB displays selection handles at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`

`{on} | off`

Object highlighted when selected.

- `on` — MATLAB indicates the selected state by drawing four edge handles and four corner handles.
- `off` — MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`Tag`

string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

For example, create an areaseries object and set the `Tag` property.

```
t = area(Y,'Tag','area1')
```

When you want to access objects of a given type, use `findobj` to find the object's handle. The following statement changes the `FaceColor` property of the object whose `Tag` is `area1`.

```
set(findobj('Tag','area1'),'FaceColor','red')
```

`Type`

string (read-only)

Type of graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For image objects, `Type` is always '`image`'.

`UIContextMenu`
handle of uicontextmenu object

Associate context menu with object. Handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object. The default value is an empty array.

`UserData`
array

User-specified data. Data you want to associate with this object (including cell arrays and structures). The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`
`{on} | off`

Visibility of object and its children.

- `on` — Object and all children of the object are visible unless the child object's `Visible` property is `off`.
- `off` — Object not displayed. However, the object still exists and you can set and query its properties.

`XData`
`[1 size(CData,2)]` by default

Control placement of image along x-axis. A vector specifying the locations of the centers of the elements `CData(1,1)` and `CData(m,n)`, where `CData` has a size of `m`-by-`n`. Element `CData(1,1)` is centered over the coordinate defined by the first

elements in `XData` and `YData`. Element `CData(m,n)` is centered over the coordinate defined by the last elements in `XData` and `YData`. The centers of the remaining elements of `CData` are evenly distributed between those two points.

The width of each `CData` element is determined by the expression:

```
(XData(2)-XData(1))/(size(CData,2)-1)
```

You can also specify a single value for `XData`. In this case, `image` centers the first element at this coordinate and centers each following element one unit apart.

`YData`

`[1 size(CData,1)]` by default

Control placement of image along y-axis. A vector specifying the locations of the centers of the elements `CData(1,1)` and `CData(m,n)`, where `CData` has a size of `m`-by-`n`. Element `CData(1,1)` is centered over the coordinate defined by the first elements in `XData` and `YData`. Element `CData(m,n)` is centered over the coordinate defined by the last elements in `XData` and `YData`. The centers of the remaining elements of `CData` are evenly distributed between those two points.

The height of each `CData` element is determined by the expression:

```
(YData(2)-YData(1))/(size(CData,1)-1)
```

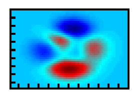
You can also specify a single value for `YData`. In this case, `image` centers the first element at this coordinate and centers each following element one unit apart.

See Also

`image`

Purpose

Scale data and display image object



Syntax

```
imagesc(C)
imagesc(x,y,C)
imagesc(...,clims)
imagesc('PropertyName',PropertyValue,...)
h = imagesc(...)
```

Description

The `imagesc` function scales image data to the full range of the current colormap and displays the image. (See "Examples" on page 1-2629 for an illustration.)

`imagesc(C)` displays `C` as an image. Each element of `C` corresponds to a rectangular area in the image. The values of the elements of `C` are indices into the current colormap that determine the color of each patch.

`imagesc(x,y,C)` displays `C` as an image and specifies the bounds of the *x*- and *y*-axis with vectors `x` and `y`. If `x(1) > x(2)` or `y(1) > y(2)`, the image is flipped left-right or up-down, respectively. If `x` and `y` are scalars, the image is translated to the specified location `(x,y)` such that the upper left corner of the image starts at `(x,y)`.

`imagesc(...,clims)` normalizes the values in `C` to the range specified by `clims` and displays `C` as an image. `clims` is a two-element vector that limits the range of data values in `C`. These values map to the full range of values in the current colormap.

`imagesc('PropertyName',PropertyValue,...)` is the low-level syntax of the `imagesc` function. It specifies only property name/property value pairs as input arguments. See Image Properties for a list of the property names and their values.

`h = imagesc(...)` returns the handle for an `image` graphics object.

Tips

`x` and `y` do not affect the elements in `C`; they only affect the annotation of the axes. If `length(x) > 2` or `length(y) > 2`, `imagesc` ignores all except the first and last elements of the respective vector.

`imagesc` creates an image with `CDataMapping` set to scaled, and sets the axes `CLim` property to the value passed in `clims`.

You cannot interactively pan or zoom outside the *x*-limits or *y*-limits of an image.

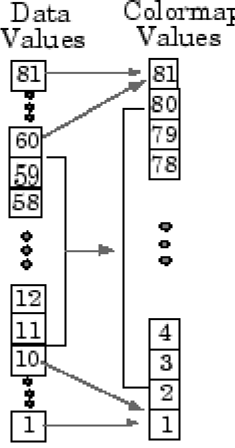
By default, `imagesc` plots the *y*-axis from lowest to highest value, top to bottom. To reverse this, type `set(gca,'YDir','normal')`. This will reverse both the *y*-axis and the image.

Examples

You can expand midrange color resolution by mapping low values to the first color and high values to the last color in the colormap by specifying color value limits (`clims`). If the size of the current colormap is 81-by-3, the statements

```
clims = [ 10 60 ]
imagesc(C,clims)
```

map the data values in C to the colormap as shown in this illustration and the code that follows:

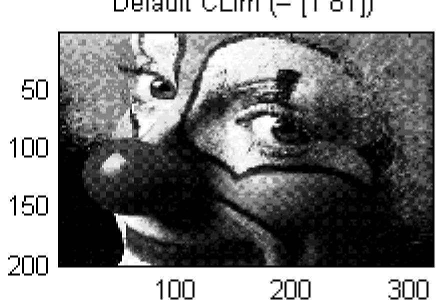


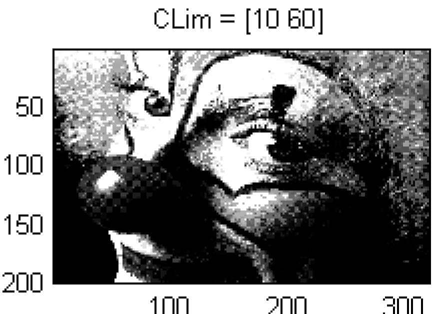
In this example, the left image maps to the `gray` colormap using the statements

```
load clown
figure
subplot(1,2,1)
imagesc(X)
colormap(gray)
axis image
title('Default CLim (= [1 81])')
```

The right image has values between 10 and 60 scaled to the full range of the `gray` colormap using the statements

```
subplot(1,2,2)
clims = [10 60];
imagesc(X,clims)
colormap(gray)
axis image
title('CLim = [10 60]')
```



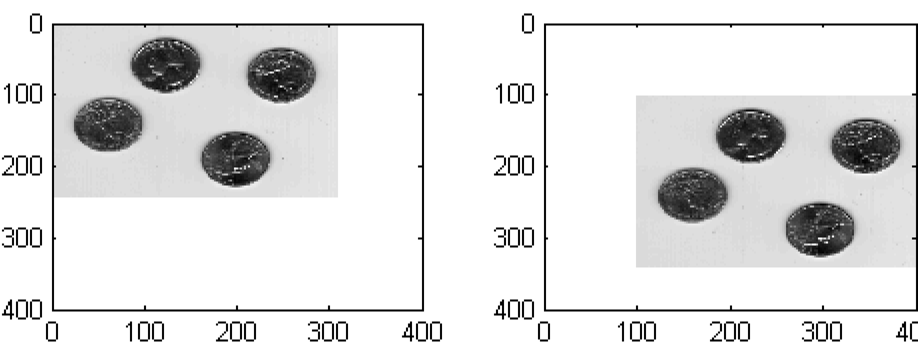


This example shows how to shift the image starting from origin to a position (100, 100),

```
i = imread('eight.tif');
```

```
figure; subplot(2,2,1); imagesc(i);
axis([0 400 0 400]);
colormap(gray);
subplot(2,2,2); imagesc(100,100,i);
axis([0 400 0 400]);
colormap(gray);
```

The figure output is as:



The top right corner of the image is now starting from (100,100) instead of the origin.

Passing vector values with the image scales the image to the specified size of 400–by–400.

```
figure; imagesc(1:400,1:400,i);
colormap(gray);
```

See Also

image | imfinfo | imread | imwrite | colorbar | colormap | pcolor | surface | surf

Purpose

Approximate indexed image by reducing number of colors

Syntax

```
[Y,newmap] = imapprox(X,map,n)
[Y,newmap] = imapprox(X,map,tol)
Y = imapprox(X,map,newmap)
Y = imapprox(...,dither_option)
```

Description

`[Y,newmap] = imapprox(X,map,n)` approximates the colors in the indexed image `X` and associated colormap `map` by using minimum variance quantization. `imapprox` returns the indexed image `Y` with colormap `newmap`, which has at most `n` colors.

`[Y,newmap] = imapprox(X,map,tol)` approximates the colors in `X` and `map` through uniform quantization. `newmap` contains at most `(floor(1/tol)+1)^3` colors. `tol` must be between 0 and 1.0.

`Y = imapprox(X,map,newmap)` approximates the colors in `map` by using colormap mapping to find the colors in `newmap` that best match the colors in `map`.

`Y = imapprox(...,dither_option)` enables or disables dithering. `dither_option` is a string that can have one of these values.

Value	Description
`{'dither'}`(default)	Dithers, if necessary, to achieve better color resolution at the expense of spatial resolution.
`'nodither'`	Maps each color in the original image to the closest color in the new map. No dithering is performed.

Class Support

The input image `X` can be of class `uint8`, `uint16`, or `double`. The output image `Y` is of class `uint8` if the length of `newmap` is less than or equal to 256. If the length of `newmap` is greater than 256, `Y` is of class `double`.

Algorithms

`imapprox` uses `rgb2ind` to create a new colormap that uses fewer colors.

Examples

Load an indexed image of a mandrill's face. Display image X using its associated colormap, map, which has 220 colors.

```
load mandrill
figure('color','k')
image(X)
colormap(map)
size(map)        % See that the color map has 220 entries

ans =
   220     3

axis off         % Remove axis ticks and numbers
axis image       % Set aspect ratio to obtain square pixels
```

Reduce the number of colors in the indexed image from 220 to only 16 colors by producing a new image, `Y`, and its associated colormap, `newmap`:

```
figure('color','k')
[Y, newmap] = imapprox(X, map, 16);
size(newmap)      % See that the new color map has 16 entries

ans =
    16     3
```

```
image(Y)
colormap(newmap)
axis off          % Remove axis ticks and numbers
axis image        % Set aspect ratio to obtain square pixels
```

See Also

cmunique | dither | rgb2ind

Purpose

Information about graphics file

Syntax

```
info = imfinfo(filename)
info = imfinfo(filename,fmt)
info = imfinfo(URL)
```

Description

`info = imfinfo(filename)` returns a structure whose fields contain information about an image in a graphics file, `filename`. The file must be in the current folder or in a folder on the MATLAB path.

The format of the file is inferred from its contents.

- If `filename` is a TIFF, HDF, ICO, GIF, or CUR file containing more than one image, then `info` is a structure array with one element for each image in the file. For example, `info(3)` would contain information about the third image in the file.

`info = imfinfo(filename,fmt)` additionally looks for a file named `filename.fmt`, if MATLAB cannot find a file named `filename`.

`info = imfinfo(URL)` returns information about the image at the specified Internet resource, `URL`.

Input Arguments

filename - Name of graphics file

string

Name of graphics file, specified as a string.

Example: `'myImage.jpg'`

Data Types
`char`

fmt - Image format

string

Image format, specified as a string. The possible values for `fmt` are contained in the MATLAB file format registry. To view of list of these formats, run the `imformats` command.

Example: `'gif'`

Data Types
`char`

URL - Image location

string

Image location, specified as a string. `URL` must include the protocol type (e.g., `http://`).

Data Types
`char`

Output Arguments

info - Information about graphics file

structure array

Information about the graphics file, returned as a structure array. The set of fields in `info` depends on the individual file and its format. This table lists the nine fields that always appear, and describes their values.

Field Name	Description	Data Type/Format?
`Filename`	Name of the file or the Internet URL specified. If the file is not in the current folder, the string contains the full path name of the file.	string
`FileModDate`	Date when the file was last modified.	string
`FileSize`	Size of the file, in bytes.	integer

Field Name	Description	Data Type/Format?
`Format`	File format, as specified by *`fmt`*. For formats with more than one possible extension (for example, JPEG and TIFF files), `imfinfo` returns the first variant in the file format registry.	string
`FormatVersion`	File format version.	string or number
`Width`	Image width, in pixels.	integer
`Height`	Image height, in pixels.	integer
`BitDepth`	Number of bits per pixel.	integer
`ColorType`	Image type. `ColorType` includes, but is not limited to, `'truecolor'` for a truecolor (RGB) image, `'grayscale'` for a grayscale intensity image, or `'indexed'` for an indexed image.	string

Additional fields returned by some file formats:

- **JPEG and TIFF only** — If `filename` contains Exchangeable Image File Format (EXIF) tags, then `info` might also contain `'DigitalCamera'` or `'GPSInfo'` (global positioning system information) fields.
- **GIF only** — `imfinfo` returns the value of the `'DelayTime'` field in hundredths of seconds.
- **JPEG2000 only** — The `info` structure contains an m-by-3 cell array, `'ChannelDefinition'`. The first column of `'ChannelDefinition'` reports a channel position as it exists in the file. The second column reports the type of channel, and the third column reports the channel mapping.

Examples

Return Information About Graphics File

Find information about the example image, ngc6543a.jpg.

```
filename = 'ngc6543a.jpg';
info = imfinfo(filename)

info =

           Filename: 'matlabroot\toolbox\matlab\demos\ngc6543a.jpg'
        FileModDate: '01-Oct-1996 16:19:44'
           FileSize: 27387
             Format: 'jpg'
      FormatVersion: ''
              Width: 600
             Height: 650
           BitDepth: 24
          ColorType: 'truecolor'
    FormatSignature: ''
    NumberOfSamples: 3
       CodingMethod: 'Huffman'
      CodingProcess: 'Sequential'
            Comment: {'CREATOR: XV Version 3.00b  Rev: 6/15/94  Quality =
'}
```

See Also

imformats | imread | imwrite

Purpose

Manage image file format registry

Syntax

```
imformats
formats = imformats
formats = imformats('fmt')
formats = imformats(format_struct)
formats = imformats('factory')
```

Description

`imformats` displays a table of information listing all the values in the MATLAB file format registry. This registry determines which file formats are supported by the `imfinfo`, `imread`, and `imwrite` functions.

`formats = imformats` returns a structure containing all the values in the MATLAB file format registry. The following tables lists the fields in the order they appear in the structure.

Field	Value
`ext`	A cell array of strings that specify filename extensions that are valid for this format
`isa`	A string specifying the name of the function that determines if a file is a certain format. This can also be a function handle.
`info`	A string specifying the name of the function that reads information about a file. This can also be a function handle.
`read`	A string specifying the name of the function that reads image data in a file. This can also be a function handle.
`write`	A string specifying the name of the function that writes MATLAB data to a file. This can also be a function handle.
`alpha`	Returns `1` if the format has an alpha channel, `0` otherwise
`description`	A text description of the file format

Note The values for the `isa`, `info`, `read`, and `write` fields must be functions on the MATLAB search path or function handles.

`formats = imformats('fmt')` searches the known formats in the MATLAB file format registry for the format associated with the filename extension `'fmt'`. If found, `imformats` returns a structure containing the characteristics and function names associated with the format. Otherwise, it returns an empty structure.

`formats = imformats(format_struct)` sets the MATLAB file format registry to the values in `format_struct`. The output structure, `formats`, contains the new registry settings.

Caution Using `imformats` to specify values in the MATLAB file format registry can result in the inability to load any image files. To return the file format registry to a working state, use `imformats` with the `'factory'` setting.

`formats = imformats('factory')` resets the MATLAB file format registry to the default format registry values. This removes any user-specified settings.

Changes to the format registry do not persist between MATLAB sessions. To have a format always available when you start MATLAB, add the appropriate `imformats` command to the MATLAB startup file, `startup.m`, located in `$MATLAB/toolbox/local` on UNIX systems, or `$MATLAB\toolbox\local` on Windows systems.

Examples

```
formats = imformats;
formats(1)

ans =

            ext: {'bmp'}
```

```
          isa: @isbmp
         info: @imbmpinfo
         read: @readbmp
        write: @writebmp
        alpha: 0
  description: 'Windows Bitmap (BMP)'
```

See Also

imfinfo | imread | imwrite | path

import

Purpose

Add package or class to current import list

Syntax

```
import package_name.class_name
import package_name.function_name
import package_name.*
import package_name.class_name1 package_name.class_name2...
import package_name1.* package_name2.*...
L = import
import
```

Description

`import` *`package_name.class_name`* adds the fully qualified class name to the current import list.

`import` *`package_name.function_name`* adds the specified package-based function.

`import` *`package_name`*`.*` adds the specified package name. Note that *`package_name`* must be followed by `.*`.

`import` *`package_name.class_name1 package_name.class_name2`*`...` adds multiple fully qualified class names.

`import` *`package_name1.* package_name2.*`*`...` adds multiple package names.

`L = import` with no input arguments returns a cell array of strings containing the current import list, without adding to it.

`import` with no input arguments displays the current import list, without adding to it.

The `import` function allows your code to refer to an imported class or function using fewer or no package prefixes.

The `import` function only affects the import list of the function within which it is used. When invoked at the command prompt, `import` uses the import list for the MATLAB command environment. If `import` is used in a script invoked from a function, it affects the import list of the function. If `import` is used in a script that is invoked from the command prompt, it affects the import list for the command environment.

The import list of a function is persistent across calls to that function and is only cleared when the function is cleared.

To clear the current import list, use the following command.

```
clear import
```

This command may only be invoked at the command prompt. Attempting to use `clear import` within a function results in an error.

Importing MATLAB Packages and Classes

You can `import` packages and classes into a MATLAB workspace (from the command line or in a function definition). For example:

```
import packagename.*
```

imports all classes and package functions so that you can reference those classes and functions by their simple names, without the package qualifier.

You can import just a single class from a package:

```
import packagename.ClassName
import Classname
```

You must still use the class name to call static methods:

```
ClassName.staticMethod()
```

For more information on how `import` works with MATLAB classes and packages, see "Importing Classes".

Tips

The `import` function allows your code to refer to an imported class by class name only, rather than with the fully qualified class name. `import` is particularly useful in streamlining calls to constructors, where most references to Java classes occur.

If you use the `import` function in a control statement, for example, `if` or `switch`, or in a function, MATLAB limits the scope of the variables

to that block of code. If you use the variables outside the function or control block, MATLAB displays an error message.

Limitations

- `import` cannot load a Java JAR package created by the MATLAB Builder™ JA product.

Examples

To add the `containers.Map` class to the current import list:

```
import containers.Map
myMap = Map('KeyType', 'char', 'ValueType', 'double');
```

To import two Java packages:

```
import java.util.Enumeration java.lang.String
s = String('hello'); % Create java.lang.String object
methods Enumeration % List java.util.Enumeration methods
```

To add the `java.awt` package:

```
import java.awt.*
f = Frame; % Create java.awt.Frame object
```

This example uses `import` in a function to call members of a .NET class in the `System.Drawing` namespace. Create the `getPrinterInfo` function:

```
function ptr = getPrinterInfo
import System.Drawing.Printing.*;
ptr = PrinterSettings;
end
```

To call the function, type:

```
NET.addAssembly('System.Drawing');
```

```
printer = getPrinterInfo;
```

See Also

clear | load | importdata

importdata

Purpose

Load data from file

Syntax

```
A = importdata(filename)
A = importdata('-pastespecial')
A = importdata( ___ ,delimiterIn)
A = importdata( ___ ,delimiterIn,headerlinesIn)
[A,delimiterOut,headerlinesOut] = importdata( ___ )
```

Description

`A = importdata(filename)` loads data into array `A`.

`A = importdata('-pastespecial')` loads data from the system clipboard rather than from a file.

`A = importdata( ___ ,delimiterIn)` interprets `delimiterIn` as the column separator in ASCII file, `filename`, or the clipboard data. You can use `delimiterIn` with any of the input arguments in the above syntaxes.

`A = importdata( ___ ,delimiterIn,headerlinesIn)` loads data from ASCII file, `filename`, or the clipboard, reading numeric data starting from line `headerlinesIn+1`.

`[A,delimiterOut,headerlinesOut] = importdata( ___ )` additionally returns the detected delimiter character for the input ASCII file in `delimiterOut` and the detected number of header lines in `headerlinesOut`, using any of the input arguments in the previous syntaxes.

Input Arguments

filename - Name and extension of file to import

string

Name and extension of the file to import, specified as a string. If `importdata` recognizes the file extension, it calls the MATLAB helper function designed to import the associated file format (such as `load`

for MAT-files or `xlsread` for spreadsheets). Otherwise, `importdata` interprets the file as a delimited ASCII file.

For ASCII files and spreadsheets, `importdata` expects to find numeric data in a rectangular form (that is, like a matrix). Text headers can appear above or to the left of the numeric data, as follows:

- Column headers or file description text at the top of the file, above the numeric data.
- Row headers to the left of the numeric data.

Example: `'myFile.jpg'`

Data Types
`char`

delimiterIn - Column separator character

string

Column separator character, specified as a string. The default character is interpreted from the file. Use `'\t'` for tab.

Example: `','`

Example: `' '`

Data Types
`char`

headerlinesIn - Number of text header lines in ASCII file

nonnegative scalar integer

Number of text header lines in the ASCII file, specified as a nonnegative scalar integer. If you do not specify `headerlinesIn`, the `importdata` function detects this value in the file.

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

Output Arguments

A - Data from the file

matrix | multidimensional array | scalar structure array

Data from the file, returned as a matrix, multidimensional array, or scalar structure array, depending on the characteristics of the file. Based on the file format of the input file, `importdata` calls a helper function to read the data. When the helper function returns more than one nonempty output, `importdata` combines the outputs into a `struct` array.

This table lists the file formats associated with helper functions that can return more than one output, and the possible fields in the structure array, `A`.

File Format	Possible Fields	Class
MAT-files	One field for each variable	Associated with each variable.

ASCII files and Spreadsheets	`data`
	`textdata`
	`colheaders`
	`rowheaders`
		For ASCII files, `data` contains a `double` array. Other fields contain `cell` arrays of strings. `textdata` includes row and column headers.
		For spreadsheets, each field contains a `struct`, with one field for each worksheet.

Images	`cdata`
	`colormap`
	`alpha`
		See `imread`.

Audio files	`data` `fs`	See `audioread`.

The MATLAB helper functions for most other supported file formats return one output. For more information about the class of each output, see the functions listed in "Supported File Formats".

If the ASCII file or spreadsheet contains either column or row headers, but not both, `importdata` returns a `colheaders` or `rowheaders` field in the output structure, where:

- `colheaders` contains only the last line of column header text. `importdata` stores all text in the `textdata` field.
- `rowheaders` is created only when the file or worksheet contains a single column of row headers.

delimiterOut - Detected column separator in the input ASCII file

string

Detected column separator in the input ASCII file, returned as a string.

headerlinesOut - Detected number of text header lines in the input ASCII file

integer

Detected number of text header lines in the input ASCII file, returned as an integer.

Examples

Import and Display an Image

Import and display the image `ngc6543a.jpg`.

```
filename = 'ngc6543a.jpg';
A = importdata(filename);
image(A);
```

The output, `A`, is class `uint8` because the helper function, `imread`, returns empty results for `colormap` and `alpha`.

Import a Text File and Specify Delimiter and Column Header

Using a text editor, create a space-delimited ASCII file with column headers called `myfile01.txt`.

```
Day1  Day2  Day3  Day4  Day5  Day6  Day7
95.01 76.21 61.54 40.57  5.79 20.28  1.53
23.11 45.65 79.19 93.55 35.29 19.87 74.68
60.68  1.85 92.18 91.69 81.32 60.38 44.51
48.60 82.14 73.82 41.03  0.99 27.22 93.18
89.13 44.47 17.63 89.36 13.89 19.88 46.60
```

Import the file, specifying the space delimiter and the single column header.

```
filename = 'myfile01.txt';
delimiterIn = ' ';
headerlinesIn = 1;
A = importdata(filename,delimiterIn,headerlinesIn);
```

View columns 3 and 5.

```
for k = [3, 5]
   disp(A.colheaders{1, k})
   disp(A.data(:, k))
   disp(' ')
end
```

```
Day3
   61.5400
   79.1900
   92.1800
   73.8200
   17.6300
```

```
Day5
    5.7900
   35.2900
   81.3200
    0.9900
   13.8900
```

Import a Text File and Return Detected Delimiter

Using a text editor, create a comma-delimited ASCII file called `myfile02.txt`.

```
1,2,3
4,5,6
7,8,9
```

Import the file, and display the output data and detected delimiter character.

```
filename = 'myfile02.txt';
[A,delimiterOut]=importdata(filename)
```

```
A =

     1     2     3
     4     5     6
     7     8     9


delimiterOut =

,
```

Import Data from Clipboard

Copy the following lines to the clipboard. Select the text, right-click, and then select **Copy**.

```
1,2,3
```

```
4,5,6
7,8,9
```

Import the clipboard data into MATLAB by typing the following.

```
A = importdata('-pastespecial')
```

```
A =

     1     2     3
     4     5     6
     7     8     9
```

Tips

- To import ASCII files with nonnumeric characters outside of column or row headers, including columns of character data or formatted dates or times, use `textscan` instead of `importdata`.

See Also

`load` | `save` | `textscan` | `xlsread` | `imread` | `uiimport`

Concepts

- "Supported File Formats"
- "Ways to Import Text Files"
- "Import Mixed Text and Numeric Data from Text Files"
- "Ways to Import Spreadsheets"
- "Process a Sequence of Files"

Purpose

Read image from graphics file

Syntax

```
A = imread(filename, fmt)
[X, map] = imread(...)
[...] = imread(filename)
[...] = imread(URL,...)
[...] = imread(...,Param1,Val1,Param2,Val2...)
```

Description

`A = imread(filename, fmt)` reads a grayscale or color image from the file specified by the string `filename`. If the file is not in the current folder, or in a folder on the MATLAB path, specify the full pathname.

The text string *`fmt`* specifies the format of the file by its standard file extension. For example, specify `'gif'` for Graphics Interchange Format files. To see a list of supported formats, with their file extensions, use the `imformats` function. If `imread` cannot find a file named `filename`, it looks for a file named `filename.fmt`.

The return value `A` is an array containing the image data. If the file contains a grayscale image, `A` is an M-by-N array. If the file contains a truecolor image, `A` is an M-by-N-by-3 array. For TIFF files containing color images that use the CMYK color space, `A` is an M-by-N-by-4 array. See TIFF in the Format-Specific Information section for more information.

The class of `A` depends on the bits-per-sample of the image data, rounded to the next byte boundary. For example, `imread` returns 24-bit color data as an array of `uint8` data because the sample size for each color component is 8 bits. See "Tips" on page 1-2656 for a discussion of bitdepths, and see "Format-Specific Information" on page 1-2656 for more detail about supported bitdepths and sample sizes for a particular format.

`[X, map] = imread(...)` reads the indexed image in `filename` into `X` and its associated colormap into `map`. Colormap values in the image file are automatically rescaled into the range `[0,1]`.

`[...]  = imread(filename)` attempts to infer the format of the file from its content.

`[...] = imread(URL,...)` reads the image from an Internet URL. The `URL` must include the protocol type (e.g., `http://`).

`[...] = imread(...,Param1,Val1,Param2,Val2...)` specifies parameters that control various characteristics of the operations for specific formats. For more information, see "Format-Specific Information" on page 1-2656.

Tips

- Bitdepth is the number of bits used to represent each image pixel. Bitdepth is calculated by multiplying the bits-per-sample with the samples-per-pixel. Thus, a format that uses 8-bits for each color component (or sample) and three samples per pixel has a bitdepth of 24. Sometimes the sample size associated with a bitdepth can be ambiguous: does a 48-bit bitdepth represent six 8-bit samples, four 12-bit samples, or three 16-bit samples? The following format-specific sections provide sample size information to avoid this ambiguity.

Format-Specific Information

The following sections provide information about the support for specific formats, listed in alphabetical order by format name. These sections include information about format-specific syntaxes, if they exist.

"BMP — Windows Bitmap" on page 1-2657

"CUR — Cursor File" on page 1-2657

"GIF — Graphics Interchange Format" on page 1-2658

"JPEG — Joint Photographic Experts Group" on page 1-2659

"JPEG 2000 — Joint Photographic Experts Group 2000" on page 1-2660

"PBM — Portable Bitmap" on page 1-2661

"PNG — Portable Network Graphics" on page 1-2662

"PPM — Portable Pixmap" on page 1-2663

"RAS — Sun Raster" on page 1-2664

"HDF4 — Hierarchical Data Format" on page 1-2659
"PCX — Windows Paintbrush" on page 1-2662
"TIFF — Tagged Image File Format" on page 1-2664
"ICO — Icon File" on page 1-2659
"PGM — Portable Graymap" on page 1-2662
"XWD — X Window Dump" on page 1-2666

BMP — Windows Bitmap

Supported Bitdepths	No Compression	RLE Compression	Output Class	Notes
1-bit	y	–	logical	
4-bit	y	y	`uint8`	
8-bit	y	y	`uint8`	
16-bit	y	–	`uint8`	1 sample/pixel
24-bit	y	–	`uint8`	3 samples/pixel
32-bit	y	–	`uint8`	3 samples/pixel (1 byte padding)

CUR — Cursor File

Supported Bitdepths	No Compression	Compression	Output Class
1-bit	y	–	logical
4-bit	y	–	`uint8`
8-bit	y	–	`uint8`

Format-specific syntaxes:

`[...] = imread(..., idx)` reads in one image from a multi-image icon or cursor file. `idx` is an integer value that specifies the order that the image appears in the file. For example, if `idx` is `3`, `imread` reads

the third image in the file. If you omit this argument, `imread` reads the first image in the file.

`[A, map, alpha] = imread(...)` returns the AND mask for the resource, which can be used to determine the transparency information. For cursor files, this mask may contain the only useful data.

Note By default, Microsoft Windows cursors are 32-by-32 pixels. MATLAB pointers must be 16-by-16. You will probably need to scale your image. If you have Image Processing Toolbox™, you can use its `imresize` function.

GIF — Graphics Interchange Format

Supported Bitdepths	Output Class
1-bit	`logical`
2-bit to 8-bit	`uint8`

Format-specific syntaxes:

`[...] = imread(..., idx)` reads in one or more frames from a multiframe (i.e., animated) GIF file. `idx` must be an integer scalar or vector of integer values. For example, if `idx` is `3`, `imread` reads the third image in the file. If `idx` is `1:5`, `imread` returns only the first five frames.

`[...] = imread(..., 'frames', idx)` is the same as the syntax above except that `idx` can be `'all'`. In this case, all the frames are read and returned in the order that they appear in the file.

Note Because of the way that GIF files are structured, all the frames must be read when a particular frame is requested. Consequently, it is much faster to specify a vector of frames or `'all'` for `idx` than to call `imread` in a loop when reading multiple frames from the same GIF file.

HDF4 — Hierarchical Data Format

Supported Bitdepths	Raster Image with colormap	Raster image without colormap	Output Class	Notes
8-bit	y	y	`uint8`	
24-bit	–	y	`uint8`	3 samples/pixel

Format-specific syntaxes:

`[...] = imread(..., ref)` reads in one image from a multi-image HDF4 file. `ref` is an integer value that specifies the reference number used to identify the image. For example, if `ref` is `12`, `imread` reads the image whose reference number is 12. (Note that in an HDF4 file the reference numbers do not necessarily correspond to the order of the images in the file. You can use `imfinfo` to match image order with reference number.) If you omit this argument, `imread` reads the first image in the file.

ICO — Icon File

See "CUR — Cursor File" on page 1-2657

JPEG — Joint Photographic Experts Group

`imread` can read any baseline JPEG image as well as JPEG images with some commonly used extensions. For information about support for JPEG 2000 files, see JPEG 2000.

Supported Bits-per-sample	Lossy Compression	Lossless Compression	Output Class	Notes
8-bit	y	y	uint8	Grayscale or RGB
12-bit	y	y	uint16	Grayscale or RGB
16-bit	–	y	uint16	Grayscale

JPEG 2000 — Joint Photographic Experts Group 2000

For information about JPEG files, see JPEG.

Note Indexed JPEG 2000 images are not supported. Only JP2 compatible color spaces are supported for JP2/JPX files. By default, all image channels are returned in the order they are stored in the file.

Supported Bits-per-sample	Lossy Compression	Lossless Compression	Output Class	Notes
1-bit	y	y	logical	Grayscale only
2- to 8-bit	y	y	uint8 or int8	Grayscale or RGB
9- to 16-bit	y	y	uint16 or int16	Grayscale or RGB

Format-specific syntaxes:

`[...] = imread(..., 'Param1', value1, 'Param2', value2, ...)` uses parameter-value pairs to control the read operation, described in the following table.

Parameter	Value
'ReductionLevel'	A non-negative integer specifying the reduction in the resolution of the image. For a reduction level L, the image resolution is reduced by a factor of 2^L. Its default value is 0 implying no reduction. The reduction level is limited by the total number of decomposition levels as specified by the 'WaveletDecompositionLevels' field in the structure returned by the imfinfo function.
'PixelRegion'	{ROWS, COLS} — The imread function returns the sub-image specified by the boundaries in ROWS and COLS. ROWS and COLS must both be two-element vectors that denote the 1-based indices [START STOP]. If 'ReductionLevel' is greater than 0, then ROWS and COLS are coordinates in the reduced-sized image.
'V79Compatible'	A logical value. If true, the image returned is transformed to grayscale or RGB, consistent with previous versions of imread (MATLAB 7.9 [R2009b] and earlier). Use this option to transform YCC images into RGB. The default is false.

PBM — Portable Bitmap

Supported Bitdepths	Raw Binary	ASCII (Plain) Encoded	Output Class
1-bit	y	y	logical

PCX — Windows Paintbrush

Supported Bitdepths	Output Class	Notes
1-bit	`logical`	Grayscale only
8-bit	`uint8`	Grayscale or indexed
24-bit	`uint8`	RGB Three 8-bit samples/pixel

PGM — Portable Graymap

Supported Bitdepths	Raw Binary	ASCII (Plain) Encoded	Output Class	Notes
8-bit	y	–	`uint8`	
16-bit	y	–	`uint16`	
Arbitrary	–	y	1-bit to 8-bit: `uint8` 9-bit to 16-bit: `uint16`	Values are scaled

PNG — Portable Network Graphics

Supported Bitdepths	Output Class	Notes
1-bit	`logical`	Grayscale
2-bit	`uint8`	Grayscale
4-bit	`uint8`	Grayscale
8-bit	`uint8`	Grayscale or Indexed
16-bit	`uint16`	Grayscale or Indexed

Supported Bitdepths	Output Class	Notes
24-bit	uint8	RGB Three 8-bit samples/pixel.
48-bit	uint16	RGB Three 16-bit samples/pixel.

Format-specific syntaxes:

`[...] = imread(...,'BackgroundColor',BG)` composites any transparent pixels in the input image against the color specified in `BG`. If `BG` is `'none'`, then no compositing is performed. If the input image is indexed, `BG` must be an integer in the range `[1,P]` where `P` is the colormap length. If the input image is grayscale, `BG` should be an integer in the range `[0,1]`. If the input image is RGB, `BG` should be a three-element vector whose values are in the range `[0,1]`. The string `'BackgroundColor'` may be abbreviated.

`[A, map, alpha] = imread(...)` returns the alpha channel if one is present; otherwise `alpha` is `[]`. Note that `map` may be empty if the file contains a grayscale or truecolor image.

If you specify the`alpha` output argument, `BG` defaults to `'none'`, if not specified. Otherwise, if the PNG file contains a background color chunk, that color is used as the default value for `BG`. If `alpha` is not used and the file does not contain a background color chunk, then the default value for `BG` is `1` for indexed images; `0` for grayscale images; and `[0 0 0]` for truecolor (RGB) images.

PPM — Portable Pixmap

Supported Bitdepths	Raw Binary	ASCII (Plain) Encoded	Output Class
Up to 16-bit	y	–	uint8
Arbitrary	–	y	

RAS — Sun™ Raster

The following table lists the supported bitdepths, compression, and output classes for RAS files.

Supported Bitdepths	Output Class	Notes
1-bit	`logical`	Bitmap
8-bit	`uint8`	Indexed
24-bit	`uint8`	RGB Three 8-bit samples/pixel
32-bit	`uint8`	RGB with Alpha Four 8-bit samples/pixel

TIFF — Tagged Image File Format

Most images supported by the TIFF specification or LibTIFF can be read by `imread`.

`imread` supports the following TIFF capabilities:

- Any number of samples-per-pixel
- CCITT group 3 and 4 FAX, Packbits, JPEG, LZW, Deflate, ThunderScan compression, and uncompressed images
- Logical, grayscale, indexed color, truecolor and hyperspectral images
- RGB, CMYK, CIELAB, ICCLAB color spaces. If the color image uses the CMYK color space, A is an M-by-N-by-4 array. To determine which color space is used, use `imfinfo` to get information about the graphics file and look at the value of the `PhotometricInterpretation` field. If a file contains CIELAB color data, `imread` converts it to ICCLAB before bringing it into the MATLAB workspace because 8- or 16-bit TIFF CIELAB-encoded values use a mixture of signed and unsigned data types that cannot be represented as a single MATLAB array.
- Data organized into tiles or scanlines

Note

- YCbCr images are converted into the RGB colorspace.
- All grayscale images are read as if black=`0`, white=largest value.
- 1-bit images are returned as class `logical`.
- CIELab images are converted into ICCLab colorspace.

The following are format-specific syntaxes for TIFF files.

`A = imread(...)` returns color data that uses the RGB, CIELAB, ICCLAB, or CMYK color spaces. If the color image uses the CMYK color space, A is an M-by-N-by-4 array.

`[...] = imread(..., 'Param1', value1, 'Param2', value2, ...)` uses parameter/value pairs to control the read operation. The following table lists the parameters you can use.

Parameter	Value
`'Index'`	Positive integer specifying which image to read. For example, if you specify the value `3`, `imread` reads the third image in the file. If you omit this argument, `imread` reads the first image in the file.
`'Info'`	Structure array returned by `imfinfo`. Note: When reading images from a multi-image TIFF file, passing the output of `imfinfo` as the value of the `'Info'` argument helps `imread` locate the images in the file more quickly.
`'PixelRegion'`	Cell array, `{Rows, Cols}`, specifying the boundaries of the region. `Rows` and `Cols` must be either two- or three-element vectors. If you specify two elements, the values denote the 1-based indices `[start stop]`. If you specify

Parameter	Value
	three elements, the values denote the 1-based indices [start increment stop], to allow image downsampling.

For copyright information, see the `libtiffcopyright.txt` file.

XWD — X Window Dump

The following table lists the supported bitdepths, compression, and output classes for XWD files.

Supported Bitdepths	ZPixmaps	XYBitmaps	XYPixmaps	Output Class
1-bit	y	–	y	`logical`
8-bit	y	–	–	`uint8`

Class Support

For most image file formats, `imread` uses 8 or fewer bits per color plane to store image pixels. The following table lists the class of the returned array for the data types used by the file formats.

Data Type Used in File	Class of Array Returned by imread
1-bit per pixel	`logical`
2- to 8-bits per color plane	`uint8`
9- to 16-bit per pixel	`uint16` (BMP, JPEG, PNG, and TIFF) For the 16-bit BMP packed format (5-6-5), MATLAB returns `uint8`

Note For indexed images, `imread` always reads the colormap into an array of class `double`, even though the image array itself may be of class `uint8` or `uint16`.

Examples

Read sample image.

```
imdata = imread('ngc6543a.jpg');
```

Read indexed image and convert it to an RGB image.

```
[cdata,map] = imread( filename )
if ~isempty( map )
   cdata = ind2rgb( cdata, map );
end
```

Read sixth image in a TIFF file.

```
[X,map] = imread('your_image.tif',6);
```

Read fourth image in an HDF4 file.

```
info = imfinfo('your_hdf_file.hdf');
[X,map] = imread('your_hdf_file.hdf',info(4).Reference);
```

Read a 24-bit PNG image and set any of its fully transparent (alpha channel) pixels to red.

```
bg = [1 0 0];
A = imread('your_image.png','BackgroundColor',bg);
```

Return the alpha channel (if any) of a PNG image.

```
[A,map,alpha] = imread('your_image.png');
```

Read an ICO image, apply a transparency mask, and then display the image.

```
[a,b,c] = imread('your_icon.ico');
% Augment colormap for background color (white).
b2 = [b; 1 1 1];
% Create new image for display.
d = ones(size(a)) * (length(b2) - 1);
% Use the AND mask to mix the background and
% foreground data on the new image
d(c == 0) = a(c == 0);
% Display new image
image(uint8(d)), colormap(b2)
```

See Also

`double` | `fread` | `image` | `imfinfo` | `imformats` | `imwrite` | `ind2rgb` | `uint8` | `uint16`

Purpose

Write image to graphics file

Syntax

```
imwrite(A,filename,fmt)
imwrite(X,map,filename,fmt)
imwrite(...,filename)
imwrite(...,Param1,Val1,Param2,Val2...)
```

Description

`imwrite(A,filename,fmt)` writes the image `A` to the file specified by `filename` in the format specified by `fmt`.

`A` can be an M-by-N (grayscale image) or M-by-N-by-3 (truecolor image) array, but it cannot be an empty array. For TIFF files, A can be an M-by-N-by-4 array containing color data that uses the CMYK color space. For GIF files, A can be an M-by-N-by-1-by-P array containing grayscale or indexed images — RGB images are not supported. For information about the class of the input array and the output image, see "Class Support" on page 1-2670.

`filename` is a string that specifies the name of the output file.

`fmt` can be any of the text strings listed in the table in "Supported Image Types" on page 1-2671. This list of supported formats is determined by the MATLAB image file format registry. See `imformats` for more information about this registry.

`imwrite(X,map,filename,fmt)` writes the indexed image in `X` and its associated colormap `map` to `filename` in the format specified by `fmt`. If `X` is of class `uint8` or `uint16`, `imwrite` writes the actual values in the array to the file. If `X` is of class `double`, `imwrite` offsets the values in the array before writing, using `uint8(X 1)`. `map` must be a valid MATLAB colormap. Note that most image file formats do not support colormaps with more than 256 entries.

When writing multiframe GIF images, `X` should be an 4-dimensional M-by-N-by-1-by-P array, where P is the number of frames to write.

`imwrite(...,filename)` writes the image to `filename`, inferring the format to use from the filename's extension. The extension must be one of the values for `fmt`, listed in "Supported Image Types" on page 1-2671.

imwrite(...,Param1,Val1,Param2,Val2...) specifies parameters that control various characteristics of the output file for HDF, JPEG, PBM, PGM, PNG, PPM, and TIFF files. For example, if you are writing a JPEG file, you can specify the quality of the output image. For the lists of parameters available for each format, see "Format-Specific Parameters" on page 1-2673.

Class Support

The input array A can be of class logical, uint8, uint16, or double. Indexed images (X) can be of class uint8, uint16, or double; the associated colormap, map, must be of class double. Input values must be full (non-sparse).

The class of the image written to the file depends on the format specified. For most formats, if the input array is of class uint8, imwrite outputs the data as 8-bit values. If the input array is of class uint16 and the format supports 16-bit data (JPEG, PNG, and TIFF), imwrite outputs the data as 16-bit values. If the format does not support 16-bit values, imwrite issues an error. Several formats, such as JPEG and PNG, support a parameter that lets you specify the bit depth of the output data.

If the input array is of class double, and the image is a grayscale or RGB color image, imwrite assumes the dynamic range is [0,1] and automatically scales the data by 255 before writing it to the file as 8-bit values.

If the input array is of class double, and the image is an indexed image, imwrite converts the indices to zero-based indices by subtracting 1 from each element, and then writes the data as uint8.

If the input array is of class logical, imwrite assumes the data is a binary image and writes it to the file with a bit depth of 1, if the format allows it. BMP, PNG, or TIFF formats accept binary images as input arrays.

Supported Image Types

This table summarizes the types of images that `imwrite` can write. The MATLAB file format registry determines which file formats are supported. See `imformats` for more information about this registry. Note that, for certain formats, `imwrite` may take additional parameters, described in "Format-Specific Parameters" on page 1-2673.

Format	Full Name	Variants
`'bmp'`	Windows Bitmap (BMP)	1-bit, 8-bit, and 24-bit uncompressed images
`'gif'`	Graphics Interchange Format (GIF)	8-bit images
`'hdf'`	Hierarchical Data Format (HDF4)	8-bit raster image data sets, with or without associated colormap, 24-bit raster image data sets; uncompressed or with RLE or JPEG compression
`'jpg'` or `'jpeg'`	Joint Photographic Experts Group (JPEG)	8-bit, 12-bit, and 16-bit Baseline JPEG images **Note** `imwrite` converts indexed images to RGB before writing data to JPEG files, because the JPEG format does not support indexed images.

Format	Full Name	Variants
`'jp2'` or `'jpx'`	"JPEG 2000 — Joint Photographic Experts Group 2000" on page 1-2676	1-bit, 8-bit, and 16-bit JPEG 2000 images
`pbm`	Portable Bitmap (PBM)	Any 1-bit PBM image, ASCII (plain) or raw (binary) encoding
`'pcx'`	Windows Paintbrush (PCX)	8-bit images
`'pgm'`	Portable Graymap (PGM)	Any standard PGM image; ASCII (plain) encoded with arbitrary color depth; raw (binary) encoded with up to 16 bits per gray value
`'png'`	Portable Network Graphics (PNG)	1-bit, 2-bit, 4-bit, 8-bit, and 16-bit grayscale images; 8-bit and 16-bit grayscale images with alpha channels; 1-bit, 2-bit, 4-bit, and 8-bit indexed images; 24-bit and 48-bit truecolor images; 24-bit and 48-bit truecolor images with alpha channels
`'pnm'`	Portable Anymap (PNM)	Any of the PPM/PGM/PBM formats, chosen automatically
`'ppm'`	Portable Pixmap (PPM)	Any standard PPM image. ASCII (plain) encoded with arbitrary color depth; raw (binary) encoded with up to 16 bits per color component

Format	Full Name	Variants
'ras'	Sun Raster (RAS)	Any RAS image, including 1-bit bitmap, 8-bit indexed, 24-bit truecolor and 32-bit truecolor with alpha
'tif' or 'tiff'	Tagged Image File Format (TIFF)	Baseline TIFF images, including 1-bit, 8-bit, 16-bit, and 24-bit uncompressed images, images with packbits compression, images with LZW compression, and images with Deflate compression; 1-bit images with CCITT 1D, Group 3, and Group 4 compression; CIELAB, ICCLAB, and CMYK images
'xwd'	X Windows Dump (XWD)	8-bit ZPixmaps

Format-Specific Parameters

GIF — Graphics Interchange Format

Format-specific parameters.

Parameter	Values
'BackgroundColor'	A scalar integer that specifies which index in the colormap should be treated as the background color for the image and is used for certain disposal methods in animated GIFs. If X is uint8 or logical, indexing starts at 0. If X is double, indexing starts at 1.
'Comment'	A string or cell array of strings containing a comment to be added to the image. For a cell array of strings, a carriage return is added after each row.
'DelayTime'	A scalar value between 0 and 655 inclusive, that specifies the delay in seconds before displaying the next image.

Parameter	Values
'DisposalMethod'	One of the following strings, which sets the disposal method of an animated GIF: 'leaveInPlace', 'restoreBG', 'restorePrevious', or 'doNotSpecify'.
'Location'	A two-element vector specifying the offset of the top left corner of the screen relative to the top left corner of the image. The first element is the offset from the top, and the second element is the offset from the left.
'LoopCount'	A finite integer between 0 and 65535 or the value Inf which specifies the number of times to repeat the animation. By default, the animation does not loop. If you specify 0, the animation will be played once. If you specify the value 1, the animation will be played twice, etc. To enable animation within Microsoft PowerPoint®, specify a value for the 'LoopCount' parameter within the range [1 65535]. Some Microsoft applications interpret the value 0 to mean do not loop at all.
'ScreenSize'	A two-element vector specifying the height and width of the frame. When used with 'Location', ScreenSize provides a way to write frames to the image that are smaller than the whole frame. 'DisposalMethod' determines the fill value used for pixels outside the frame.
'TransparentColor'	A scalar integer. This value specifies which index in the colormap should be treated as the transparent color for the image. If X (the image) is uint8 or logical, indexing starts at 0. If X is double, indexing starts at 1.
'WriteMode'	'overwrite' (the default) or 'append'. In append mode, imwrite adds a single frame to the existing file.

HDF4 — Hierarchical Data Format

Format-specific parameters.

Parameter	Values
'Compression'	'none' (the default) 'jpeg'(valid only for grayscale and RGB images) 'rle' (valid only for grayscale and indexed images)
'Quality'	A number between 0 and 100; this parameter applies only if 'Compression' is 'jpeg'. Higher numbers mean higher *quality* (less image degradation due to compression), but the resulting file size is larger. The default value is 75.
'WriteMode'	'overwrite' (the default) 'append'

JPEG — Joint Photographic Experts Group

Format-specific parameters. For information about support for JPEG 2000 files, see "JPEG 2000 — Joint Photographic Experts Group 2000" on page 1-2676.

Parameter	Values	Default
'Bitdepth'	A scalar value indicating desired bitdepth; for grayscale images this can be 8, 12, or 16; for color images this can be 8 or 12.	8 (grayscale) and 8 bit per plane for color images
'Comment'	A column vector cell array of strings or a character matrix. imwrite writes each row of input as a comment in the JPEG file.	Empty

Parameter	Values	Default
'Mode'	Specifies the type of compression used:'lossy' or 'lossless'	'lossy'
'Quality'	A number between 0 and 100; higher numbers mean higher quality (less image degradation due to compression), but the resulting file size is larger.	75

JPEG 2000 — Joint Photographic Experts Group 2000

Format-specific parameters. For information about support for JPEG files, see "JPEG — Joint Photographic Experts Group" on page 1-2675.

Parameter	Values	Default
'Comment'	A column vector cell array of strings or a character matrix. imwrite writes each row of input as a comment in the JPEG file.	Empty
'CompressionRatio'	A real value greater than 1 specifying the target compression ratio which is defined as the ratio of input image size to the output compressed size. For example, a value of 2.0 implies that the output image size will be half of the input image size or less. A higher value implies a smaller file size and reduced image quality. This is valid only with 'lossy' mode. Note that the compression ratio doesn't take into account the header size, and hence in some cases the output file size can be larger than expected.	
'Mode'	Specifies the type of compression used:'lossy' or 'lossless'	'lossy'

Parameter	Values	Default
'ProgressionOrder'	One of the following text strings: 'LRCP', 'RLCP', 'RPCL', 'PCRL' or 'CPRL'. The characters in these text strings represent the following: L=layer, R=resolution, C=component and P=position. The first character refers to the index which progresses most slowly, while the last refers to the index which progresses most quickly.	'LRCP'
'QualityLayers'	A positive integer (not exceeding 20) specifying the number of quality layers.	1
'ReductionLevels'	A positive integer (not exceeding 8) specifying the number of reduction levels or the wavelet decomposition levels	
'TileSize'	A 2-element vector specifying tile height and tile width. The minimum tile size that can be specified is [128 128].	tile size is same as the image size

PBM-, PGM-, and PPM-Specific Parameters

This table describes the available parameters for PBM, PGM, and PPM files.

Parameter	Values	Default
'Encoding'	One of these strings: 'ASCII' for plain encoding 'rawbits' for binary encoding	'rawbits'
'MaxValue'	A scalar indicating the maximum gray or color value. Available only for PGM and PPM files. For PBM files, this value is always 1.	Default is 65535 if image array is 'uint16'; 255 otherwise.

PNG-Specific Parameters

The following table lists the available parameters for PNG files, in alphabetical order. In addition to these PNG parameters, you can use any parameter name that satisfies the PNG specification for keywords; that is, uses only printable characters, contains 80 or fewer characters, and no contains no leading or trailing spaces. The value corresponding to these user-specified parameters must be a string that contains no control characters other than linefeed.

Parameter	Values
`'Alpha'`	A matrix specifying the transparency of each pixel individually. The row and column dimensions must be the same as the data array; they can be `uint8`, `uint16`, or `double`, in which case the values should be in the range `[0,1]`.
`'Author'`	A string
`'Background'`	The value specifies background color to be used when compositing transparent pixels. For indexed images: an integer in the range `[1,P]`, where `P` is the colormap length. For grayscale images: a scalar in the range `[0,1]`. For truecolor images: a three-element vector in the range `[0,1]`.
`'bitdepth'`	A scalar value indicating desired bit depth. For grayscale images this can be 1, 2, 4, 8, or 16. For grayscale images with an alpha channel this can be 8 or 16. For indexed images this can be 1, 2, 4, or 8. For truecolor images with or without an alpha channel this can be 8 or 16. By default, `imwrite` uses 8 bits per pixel, if image is double or `uint8`; 16 bits per pixel if image is `uint16`; 1 bit per pixel if image is `logical`.

Parameter	Values
'Chromaticities'	An eight-element vector [wx wy rx ry gx gy bx by] that specifies the reference white point and the primary chromaticities
'Comment'	A string
'Copyright'	A string
'CreationTime'	A string
'Description'	A string
'Disclaimer'	A string
'Gamma'	A nonnegative scalar indicating the file gamma
'ImageModTime'	A MATLAB serial date number (see the datenum function) or a string convertible to a date vector via the datevec function. Values should be in Coordinated Universal Time (UTC).
'InterlaceType'	Either 'none' (the default) or 'adam7'
'ResolutionUnit'	Either 'unknown' or 'meter'
'SignificantBits'	A scalar or vector indicating how many bits in the data array should be regarded as significant; values must be in the range [1,BitDepth]. For indexed images: a three-element vector. For grayscale images: a scalar. For grayscale images with an alpha channel: a two-element vector. For truecolor images: a three-element vector. For truecolor images with an alpha channel: a four-element vector.
'Software'	A string
'Source'	A string

Parameter	Values
'Transparency'	This value is used to indicate transparency information only when no alpha channel is used. Set to the value that indicates which pixels should be considered transparent. (If the image uses a colormap, this value represents an index number to the colormap.) For indexed images: a Q-element vector in the range [0,1], where Q is no larger than the colormap length and each value indicates the transparency associated with the corresponding colormap entry. In most cases, Q = 1. For grayscale images: a scalar in the range [0,1]. The value indicates the grayscale color to be considered transparent. For truecolor images: a three-element vector in the range [0,1]. The value indicates the truecolor color to be considered transparent. **Note** You cannot specify 'Transparency' and 'Alpha' at the same time.
'Warning'	A string
'XResolution'	A scalar indicating the number of pixels/unit in the horizontal direction
'YResolution'	A scalar indicating the number of pixels/unit in the vertical direction

RAS-Specific Parameters

This table describes the available parameters for RAS files.

Parameter	Values	Default
'Alpha'	A matrix specifying the transparency of each pixel individually; the row and column dimensions must be the same as the data array; can be uint8, uint16, or double. Can only be used with truecolor images.	Empty matrix ([])
'Type'	One of these strings: 'standard' (uncompressed, b-g-r color order with truecolor images) 'rgb' (like 'standard', but uses r-g-b color order for truecolor images) 'rle' (run-length encoding of 1-bit and 8-bit images)	'standard'

TIFF-Specific Parameters

This table describes the available parameters for TIFF files.

Parameter	Values
'ColorSpace'	Specifies the color space used to represent the color data. 'rgb' 'cielab' 'icclab' [1] (default is 'rgb'). Note: To use the CMYK color space in a TIFF file, do not use the 'ColorSpace' parameter. It is sufficient to specify an M-by-N-by-4 input array.
'Compression'	'none' 'packbits' (default for non-binary images) 'lzw' 'deflate' 'jpeg' 'ccitt' (binary images only; default) 'fax3' (binary images only) 'fax4' (binary images only) Note: 'jpeg' is a lossy compression scheme; other compression modes are lossless. Also, if you specify 'jpeg' compression, you

Parameter	Values
	must specify the 'RowsPerStrip' parameter and the value must be a multiple of 8.
'Description'	Any string; fills in the ImageDescription field returned by imfinfo. By default, the field is empty.
'Resolution'	A two-element vector containing the XResolution and YResolution, or a scalar indicating both resolutions. The default value is 72.
'RowsPerStrip'	A scalar value specifying the number of rows to include in each strip. The default will be such that each strip is about 8K bytes. Note: You must specify the 'RowsPerStrip' parameter if you specify 'jpeg' compression. The value must be a multiple of 8.
'WriteMode'	'overwrite' (default) 'append'

[1]imwrite can write color image data that uses the *L*a*b** color space to TIFF files. The 1976 CIE *L*a*b** specification defines numeric values that represent luminance (*L**) and chrominance (*a** and *b**) information. To store *L*a*b** color data in a TIFF file, the values must be encoded to fit into either 8-bit or 16-bit storage. imwrite can store *L*a*b** color data in a TIFF file using 8-bit and 16-bit encodings defined by the TIFF specification, called the CIELAB encodings, or using 8-bit and 16-bit encodings defined by the International Color Consortium, called ICCLAB encodings.

The output class and encoding used by imwrite depends on the class of the input array and the value of the TIFF-specific ColorSpace parameter. The following table explains these options. (The 8-bit and 16-bit CIELAB encodings cannot be input arrays because they use a mixture of signed and unsigned values and cannot be represented as a single MATLAB array.)

Input Class and Encoding	ColorSpace Parameter Value	Output Class and Encoding
8-bit ICCLAB Represents values as integers in the range [0 255]. L^* values are multiplied by `255/100`. `128` is added to both the a^* and b^* values.	'icclab'	8-bit ICCLAB
	'cielab'	8-bit CIELAB
16-bit ICCLAB Represents the values as integers in the range [0, 65280]. L^* values are multiplied by `65280/100`. `32768` is added to both the a^* and b^* values, which are represented as integers in the range [0,65535].	'icclab'	16-bit ICCLAB
	'cielab'	16-bit CIELAB

Input Class and Encoding	ColorSpace Parameter Value	Output Class and Encoding
Double-precision 1976 CIE $L^*a^*b^*$ values L^* is in the dynamic range [0, 100]. a^* and b^* can take any value. Setting a^* and b^* to 0 (zero) produces a neutral color (gray).	'icclab'	8-bit ICCLAB
	'cielab'	8-bit CIELAB

For copyright information, see the `libtiffcopyright.txt` file.

Examples

This example appends an indexed image `X` and its colormap `map` to an existing uncompressed multipage HDF4 file.

```
imwrite(X,map,'your_hdf_file.hdf','Compression','none',...
'WriteMode','append')
```

See Also

`fwrite` | `getframe` | `imfinfo` | `imformats` | `imread`

Purpose

(Will be removed) Incenters of specified simplices

Note `incenters(TriRep)` will be removed in a future release. Use `incenter(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
IC = incenters(TR,SI)
[IC RIC] = incenters(TR, SI)
```

Description

`IC = incenters(TR,SI)` returns the coordinates of the incenter of each specified simplex `SI`.

`[IC RIC] = incenters(TR, SI)` returns the incenters and the corresponding radius of the inscribed circle/sphere.

Input Arguments

`TR`	Triangulation representation.
`SI`	Column vector of simplex indices that index into the triangulation matrix `TR.Triangulation`. If `SI` is not specified the incenter information for the entire triangulation is returned, where the incenter associated with simplex `i` is the `i`'th row of `IC`.

Output Arguments

`IC`	`m`-by-`n` matrix, where `m = length(SI)`, the number of specified simplices, and `n` is the dimension of the space where the triangulation resides. Each row `IC(i,:)` represents the coordinates of the incenter of simplex `SI(i)`.
`RIC`	Vector of length `length(SI)`, the number of specified simplices.

Definitions

A simplex is a triangle/tetrahedron or higher-dimensional equivalent.

Examples

Example 1

Load a 3-D triangulation:

```
load tetmesh
```

Use `TriRep` to compute the incenters of the first five tetrahedra.

```
trep = TriRep(tet, X)
ic = incenters(trep, [1:5]')
```

Example 2

Query a 2-D triangulation created with `DelaunayTri`.

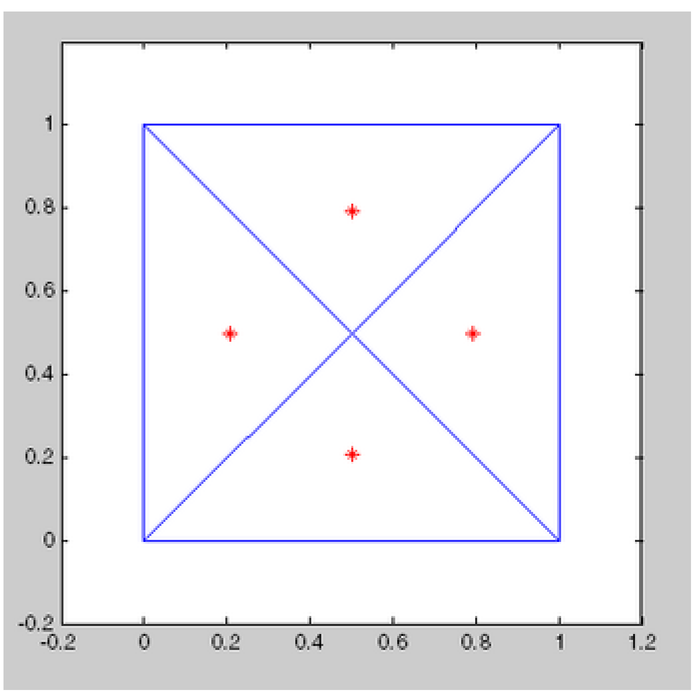
```
x = [0 1 1 0 0.5]';
y = [0 0 1 1 0.5]';
dt = DelaunayTri(x,y);
```

Compute incenters of the triangles:

```
ic = incenters(dt);
```

Plot the triangles and incenters:

```
triplot(dt);
axis equal;
axis([-0.2 1.2 -0.2 1.2]);
hold on;
plot(ic(:,1),ic(:,2),'*r');
hold off;
```



See Also

`delaunayTriangulation` | `circumcenter` | `triangulation`

DelaunayTri.inOutStatus

Purpose

(Will be removed) Status of triangles in 2-D constrained Delaunay triangulation

---

Note `inOutStatus(DelaunayTri)` will be removed in a future release. Use `isInterior(delaunayTriangulation)` instead.

`DelaunayTri` will be removed in a future release. Use `delaunayTriangulation` instead.

Syntax

```
IN = inOutStatus(DT)
```

Description

`IN = inOutStatus(DT)` returns the in/out status of the triangles in a 2-D constrained Delaunay triangulation of a geometric domain. Given a Delaunay triangulation that has a set of constrained edges that define a bounded geometric domain. The `i`'th triangle in the triangulation is classified as inside the domain if `IN(i) = 1` and outside otherwise.

Note `inOutStatus` is only relevant for 2-D constrained Delaunay triangulations where the imposed edge constraints bound a closed geometric domain.

Input Arguments

`DT`	Delaunay triangulation.

Output Arguments

`IN`	Logical array of length equal to the number of triangles in the triangulation. The constrained edges in the triangulation define the boundaries of a valid geometric domain.

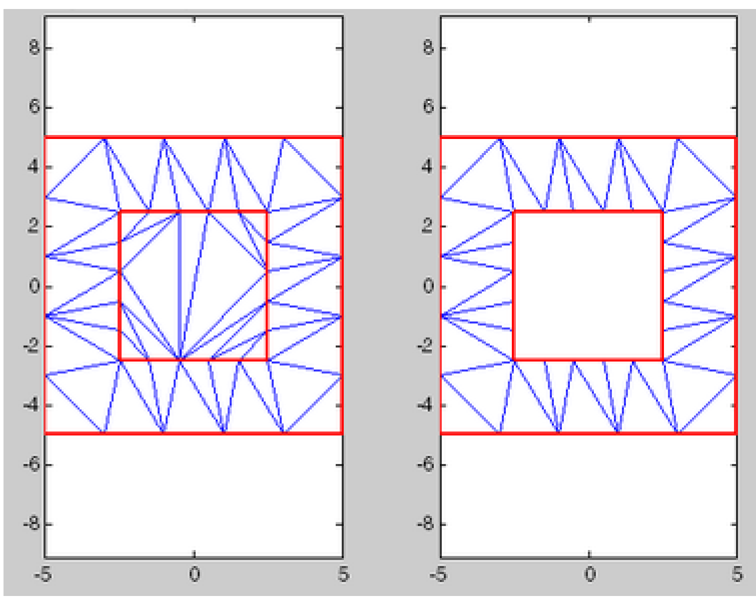
Examples

Create a geometric domain that consists of a square with a square hole:

```
outerprofile = [-5 -5; -3 -5; -1 -5; 1 -5; 3 -5; ...
 5 -5; 5 -3; 5 -1; 5 1; 5 3;...
 5 5; 3 5; 1 5; -1 5; -3 5; ...
 -5 5; -5 3; -5 1; -5 -1; -5 -3; ];
innerprofile = outerprofile.*0.5;
profile = [outerprofile; innerprofile];
outercons = [(1:19)' (2:20)'; 20 1;];
innercons = [(21:39)' (22:40)'; 40 21];
edgeconstraints = [outercons; innercons];
```

Create a constrained Delaunay triangulation of the domain:

```
dt = DelaunayTri(profile, edgeconstraints)
subplot(1,2,1);
triplot(dt);
hold on;
plot(dt.X(outercons',1), dt.X(outercons',2), ...
     '-r', 'LineWidth', 2);
plot(dt.X(innercons',1), dt.X(innercons',2), ...
     '-r', 'LineWidth', 2);
axis equal;
% Plot showing interior and exterior
% triangles with respect to the domain.
hold off;
subplot(1,2,2);
inside = inOutStatus(dt);
triplot(dt(inside, :), dt.X(:,1), dt.X(:,2));
hold on;
plot(dt.X(outercons',1), dt.X(outercons',2), ...
     '-r', 'LineWidth', 2);
plot(dt.X(innercons',1), dt.X(innercons',2), ...
     '-r', 'LineWidth', 2);
axis equal;
% Plot showing interior triangles only
hold off;
```



See Also

`delaunayTriangulation` | `triangulation` | `isInterior`

Purpose

Convert indexed image to RGB image

Syntax

```
RGB = ind2rgb(X,map)
```

Description

`RGB = ind2rgb(X,map)` converts the matrix `X` and corresponding colormap `map` to RGB (truecolor) format.

Class Support

`X` can be of class `uint8`, `uint16`, `single`, or `double`. RGB is an m-by-n-by-3 array of class `double`.

See Also

`image` | `imread`

ind2sub

Purpose

Subscripts from linear index

Syntax

```
[I,J] = ind2sub(siz,IND)
[I1,I2,I3,...,In] = ind2sub(siz,IND)
```

Description

The `ind2sub` command determines the equivalent subscript values corresponding to a single index into an array.

`[I,J] = ind2sub(siz,IND)` returns the matrices `I` and `J` containing the equivalent row and column subscripts corresponding to each linear index in the matrix `IND` for a matrix of size `siz`. `siz` is a vector with `ndim(A)` elements (in this case, 2), where `siz(1)` is the number of rows and `siz(2)` is the number of columns.

Note For matrices, `[I,J] = ind2sub(size(A),find(A>5))` returns the same values as `[I,J] = find(A>5)`.

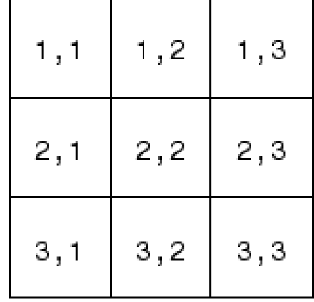
`[I1,I2,I3,...,In] = ind2sub(siz,IND)` returns `n` subscript arrays `I1`,`I2`,`...`,`In` containing the equivalent multidimensional array subscripts equivalent to `IND` for an array of size `siz`. `siz` is an `n`-element vector that specifies the size of each array dimension.

Examples

Example 1 — Two-Dimensional Matrices

The mapping from linear indexes to subscript equivalents for a 3-by-3 matrix is

1	4	7
2	5	8
3	6	9

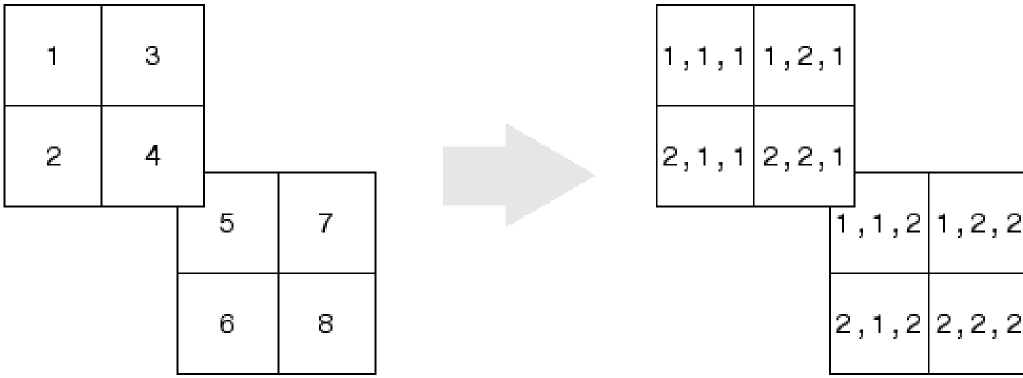


This code determines the row and column subscripts in a 3-by-3 matrix, of elements with linear indices 3, 4, 5, 6.

```
IND = [3 4 5 6]
s = [3,3];
[I,J] = ind2sub(s,IND)

I =
     3     1     2     3

J =
     1     2     2     2
```

Example 2 — Three-Dimensional Matrices

The mapping from linear indexes to subscript equivalents for a 2-by-2-by-2 array is



This code determines the subscript equivalents in a 2-by-2-by-2 array, of elements whose linear indices 3, 4, 5, 6 are specified in the `IND` matrix.

```
IND = [3 4;5 6];
s = [2,2,2];
[I,J,K] = ind2sub(s,IND)

I =
     1     2
     1     2

J =
     2     2
     1     1

K =
     1     1
     2     2
```

Example 3 — Effects of Returning Fewer Outputs

When calling `ind2sub` for an N-dimensional matrix, you would typically supply N output arguments in the call: one for each dimension of the matrix. This example shows what happens when you return three, two, and one output when calling `ind2sub` on a 3-dimensional matrix.

The matrix is 2-by-2-by-2 and the linear indices are 1 through 8:

```
dims = [2 2 2];
indices = [1 2 3 4 5 6 7 8];
```

The 3-output call to `ind2sub` returns the expected subscripts for the 2-by-2-by-2 matrix:

```
[rowsub colsub pagsub] = ind2sub(dims, indices)
rowsub =
     1     2     1     2     1     2     1     2
colsub =
     1     1     2     2     1     1     2     2
pagsub =
     1     1     1     1     2     2     2     2
```

If you specify only two outputs (row and column), `ind2sub` still returns a subscript for each specified index, but drops the third dimension from the matrix, returning subscripts for a 2-dimensional, 2-by-4 matrix instead:

```
[rowsub colsub] = ind2sub(dims, indices)
rowsub =
     1     2     1     2     1     2     1     2
colsub =
     1     1     2     2     3     3     4     4
```

If you specify one output (row), `ind2sub` drops both the second and third dimensions from the matrix, and returns subscripts for a 1-dimensional, 1-by-8 matrix instead:

```
[rowsub] = ind2sub(dims, indices)
rowsub =
     1     2     3     4     5     6     7     8
```

See Also

`find` | `size` | `sub2ind`

Inf

Purpose

Infinity

Syntax

```
Inf
Inf('double')
Inf('single')
Inf(n)
Inf(m,n)
Inf(m,n,p,...)
Inf(...,classname)
Inf(...,'like',p)
```

Description

`Inf` returns the IEEE arithmetic representation for positive infinity. Infinity results from operations like division by zero and overflow, which lead to results too large to represent as conventional floating-point values.

`Inf('double')` is the same as `Inf` with no inputs.

`Inf('single')` is the single precision representation of `Inf`.

`Inf(n)` is an `n`-by-`n` matrix of `Inf`s.

`Inf(m,n)` or `inf([m,n])` is an `m`-by-`n` matrix of `Inf`s.

`Inf(m,n,p,...)` or `Inf([m,n,p,...])` is an `m`-by-`n`-by-`p`-by-... array of `Inf`s.

Note The size inputs `m`, `n`, `p`, `...` should be nonnegative integers. Negative integers are treated as 0.

`Inf(...,classname)` is an array of `Inf`s of class specified by the string `classname`. `classname` can be either `'single'` or `'double'`.

`Inf(...,'like',p)` is an array of `Inf`s of the same data type, sparsity, and complexity (real or complex) as the single or double precision numeric variable `p`.

Examples

`1/0`, `1.e1000`, `2^2000`, and `exp(1000)` all produce `Inf`.

`log(0)` produces `-Inf`.

`Inf-Inf` and `Inf/Inf` both produce `NaN` (Not-a-Number).

See Also

`isinf` | `NaN`

inferiorto

Purpose

Specify inferior class relationship

Syntax

```
inferiorto('class1','class2',...)
```

Description

`inferiorto('class1','class2',...)` establishes that the class invoking this function in its constructor has lower precedence than the classes in the argument list. MATLAB uses this precedence to determines which method or function MATLAB calls in any given situation.

Use this function only from a constructor that calls the `class` function to create objects (classes defined before MATLAB 7.6).

Examples

Specify class precedence.

Suppose `a` is an object of class `class_a`, `b` is an object of class `class_b`, and `c` is an object of class `class_c`. Suppose the constructor method of `class_c` contains the statement:

```
inferiorto('class_a')
```

This function call establishes `class_a` as taking precedence over `class_c` for function dispatching. Therefore, either of the following two statements:

```
e = fun(a,c);
e = fun(c,a);
```

Invoke `class_a/fun`.

If you call a function with two objects having an unspecified relationship, the two objects have equal precedence. In this case, MATLAB calls the method of the left-most object. So `fun(b, c)` calls `class_b/fun`, while `fun(c, b)` calls `class_c/fun`.

See Also

`superiorto`

Purpose

Information about contacting MathWorks

Note `info` will be removed in a future release.

Syntax

```
info
```

Description

`info` displays in the Command Window, information about contacting MathWorks.

See Also

`help` | `version`

inline

Purpose

Construct inline object

Compatibility

`inline` will be removed in a future release. Use "Anonymous Functions" instead.

Syntax

```
inline(expr)
inline(expr,arg1,arg2,...)
inline(expr,n)
```

Description

`inline(expr)` constructs an inline function object from the MATLAB expression contained in the string `expr`. The input argument to the inline function is automatically determined by searching `expr` for an isolated lower case alphabetic character, other than `i` or `j`, that is not part of a word formed from several alphabetic characters. If no such character exists, `x` is used. If the character is not unique, the one closest to `x` is used. If two characters are found, the one later in the alphabet is chosen.

`inline(expr,arg1,arg2,...)` constructs an inline function whose input arguments are specified by the strings `arg1`, `arg2`,.... Multicharacter symbol names may be used.

`inline(expr,n)` where `n` is a scalar, constructs an inline function whose input arguments are `x`, `P1`, `P2`, ...

Tips

Three commands related to `inline` allow you to examine an inline function object and determine how it was created.

`char(fun)` converts the inline function into a character array. This is identical to `formula(fun)`.

`argnames(fun)` returns the names of the input arguments of the inline object `fun` as a cell array of strings.

`formula(fun)` returns the formula for the inline object `fun`.

A fourth command `vectorize(fun)` inserts a `.` before any `^`, `*` or `/` in the formula for `fun`. The result is a vectorized version of the inline function.

Examples

Example 1

This example creates a simple inline function to square a number.

```
g = inline('t^2')
g =

     Inline function:
     g(t) = t^2
```

You can convert the result to a string using the `char` function.

```
char(g)

ans =

t^2
```

Example 2

This example creates an inline function to represent the formula $f = 3\sin(2x^2)$. The resulting inline function can be evaluated with the `argnames` and `formula` functions.

```
f = inline('3*sin(2*x.^2)')

f =
     Inline function:
     f(x) = 3*sin(2*x.^2)

argnames(f)

ans =
    'x'

formula(f)
ans =

3*sin(2*x.^2)
```

Example 3

This call to `inline` defines the function `f` to be dependent on two variables, `alpha` and `x`:

```
f = inline('sin(alpha*x)')

f =
     Inline function:
     f(alpha,x) = sin(alpha*x)
```

If `inline` does not return the desired function variables or if the function variables are in the wrong order, you can specify the desired variables explicitly with the `inline` argument list.

```
g = inline('sin(alpha*x)','x','alpha')

g =

     Inline function:
     g(x,alpha) = sin(alpha*x)
```

Purpose

Names of functions, MEX-files, classes in memory

Syntax

```
M = inmem
[M,X] = inmem
[M,X,C] = inmem
[...] = inmem('-completenames')
```

Description

`M = inmem` returns a cell array of strings containing the names of the functions that are currently loaded.

`[M,X] = inmem` returns an additional cell array `X` containing the names of the MEX-files that are currently loaded.

`[M,X,C] = inmem` also returns a cell array `C` containing the names of the classes that are currently loaded.

`[...]  = inmem('-completenames')` returns not only the names of the currently loaded function and MEX-files, but the path and filename extension for each as well. No additional information is returned for loaded classes.

Examples

Functions in Memory

List the functions that remain in memory after calling the `magic` function.

```
clear all
magic(10);

M = inmem


M =
    'workspacefunc'
    'magic'
```

The function list includes `magic` and additional functions that are in memory in your current session.

MEX-Files in Memory

Call a sample MEX-function named `arrayProduct`, and then verify that the MEX-function is in memory.

```
clear all
sampleFolder = fullfile(matlabroot,'extern','examples','mex');
addpath(sampleFolder);

s = 5;
A = [1.5, 2, 9];
B = arrayProduct(s,A);

[M,X] = inmem('-completenames');
X
```

```
X =
    'matlabroot\extern\examples\mex\arrayProduct.mexw64'
```

See Also

`clear`

Purpose

Points inside polygonal region

Syntax

```
IN = inpolygon(X,Y,xv,yv)
[IN ON] = inpolygon(X,Y,xv,yv)
```

Description

`IN = inpolygon(X,Y,xv,yv)` returns a matrix `IN` the same size as `X` and `Y`. Each element of `IN` is assigned the value 1 or 0 depending on whether the point `(X(p,q),Y(p,q))` is inside the polygonal region whose vertices are specified by the vectors `xv` and `yv`. In particular:

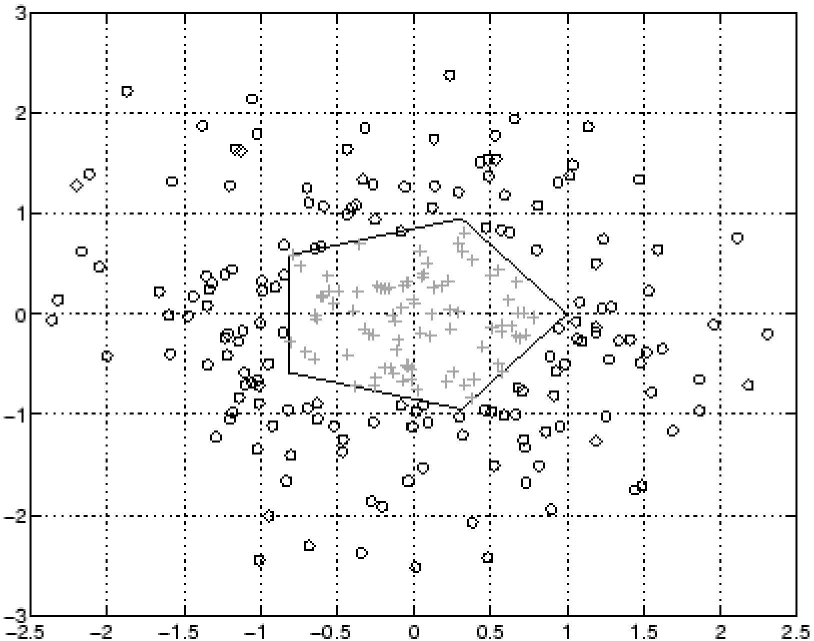
`IN(p,q) = 1`	If `(X(p,q),Y(p,q))` is inside the polygonal region or on the polygon boundary
`IN(p,q) = 0`	If `(X(p,q),Y(p,q))` is outside the polygonal region

`[IN ON] = inpolygon(X,Y,xv,yv)` returns a second matrix `ON` the same size as `X` and `Y`. Each element of `ON` is assigned the value 1 or 0 depending on whether the point `(X(p,q),Y(p,q))` is on the boundary of the polygonal region whose vertices are specified by the vectors `xv` and `yv`. In particular:

`ON(p,q) = 1`	If `(X(p,q),Y(p,q))` is on the polygon boundary
`ON(p,q) = 0`	If `(X(p,q),Y(p,q))` is inside or outside the polygon boundary

Examples

```
L = linspace(0,2.*pi,6); xv = cos(L)';yv = sin(L)';
xv = [xv ; xv(1)]; yv = [yv ; yv(1)];
x = randn(250,1); y = randn(250,1);
in = inpolygon(x,y,xv,yv);
plot(xv,yv,x(in),y(in),'r+',x(~in),y(~in),'bo')
```


3
2
1
0
−1
−2
−3
−2.5
−2
−1.5
−1
−0.5
0
0.5
1
1.5
2
2.5

Purpose

Request user input

Syntax

```
result = input(prompt)
str = input(prompt,'s')
```

Description

`result = input(prompt)` displays the `prompt` string on the screen, waits for input from the keyboard, evaluates any expressions in the input, and returns the result. To evaluate expressions, the `input` function can use variables in the current workspace.

- If you press the **Return** key without entering anything, then `input` returns an empty matrix.
- If you enter an invalid expression at the prompt, then MATLAB displays the relevant error message, and then redisplays the prompt.

`str = input(prompt,'s')` returns the entered text as a MATLAB string, without evaluating expressions.

Input Arguments

prompt - Query that requests input

string

Query that requests input, specified as a string.

To create a prompt that spans several lines, use `'\n'` to indicate each new line. To include a backslash (`'\'`) in the prompt, use `'\\'`.

Data Types

`char`

Output Arguments

result - Result calculated from input

array

Result calculated from input, returned as an array. The type and dimensions of the array depend upon the response to the prompt.

str - Exact text of input

string

Exact text of input, returned as a string.

Data Types
`char`

Examples

Numeric or Evaluated Input

Request a numeric input, and then multiply the input by 10.

```
prompt = 'What is the original value? ';
result = input(prompt)
largernum = result * 10
```

At the prompt, enter a numeric value or array, such as `42`.

```
result =
    42

largernum =
   420
```

The `input` function also accepts expressions. For example, rerun the code.

```
prompt = 'What is the original value? ';
result = input(prompt)
largernum = result * 10
```

At the prompt, enter `magic(3)`.

```
result =
     8     1     6
     3     5     7
     4     9     2

largernum =
    80    10    60
    30    50    70
    40    90    20
```

The expression does not need to return a numeric result. For example:

```
prompt = 'What color is the sun? ';
result = input(prompt)
```

At the prompt, type `upper('yellow')`.

```
result =
YELLOW
```

Unprocessed Text Input

Request a simple text response that requires no evaluation.

```
prompt = 'Do you want more? Y/N [Y]: ';
str = input(prompt,'s');
if isempty(str)
    str = 'Y';
end
```

The `input` function returns the text exactly as typed. If the input is empty, this code assigns a default value, `'Y'`, to the output string, `str`.

See Also

`keyboard` | `inputdlg` | `menu` | `ginput` | `uicontrol`

Purpose

Create and open input dialog box

Syntax

```
answer = inputdlg(prompt)
answer = inputdlg(prompt,dlg_title)
answer = inputdlg(prompt,dlg_title,num_lines)
answer = inputdlg(prompt,dlg_title,num_lines,defAns)
answer = inputdlg(prompt,dlg_title,num_lines,defAns,options)
```

Description

`answer = inputdlg(prompt)` creates a modal dialog box and returns user input for multiple prompts in the cell array. `prompt` is a cell array containing prompt strings.

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

`answer = inputdlg(prompt,dlg_title)` `dlg_title` specifies a title for the dialog box.

`answer = inputdlg(prompt,dlg_title,num_lines)` `num_lines` specifies the number of lines for each user-entered value. `num_lines` can be a scalar, column vector, or a m x 2 array.

- If `num_lines` is a scalar, it applies to all prompts.
- If `num_lines` is a column vector, each element specifies the number of lines of input for a prompt.
- If `num_lines` is an array, it must be size m-by-2, where m is the number of prompts on the dialog box. Each row refers to a prompt. The first column specifies the number of lines of input for a prompt. The second column specifies the width of the field in characters.

`answer = inputdlg(prompt,dlg_title,num_lines,defAns)` `defAns` specifies the default value to display for each prompt. `defAns` must contain the same number of elements as `prompt` and all elements must be strings.

answer =
inputdlg(prompt,dlg_title,num_lines,defAns,options) If options is the string 'on', the dialog is made resizable in the horizontal direction. If options is a structure, the fields shown in the following table are recognized:

Field	Description
Resize	Can be 'on' or 'off' (default). If 'on', the window is resizable horizontally.
WindowStyle	Can be either 'normal' or 'modal' (default).
Interpreter	Can be either 'none' (default) or 'tex'. If the value is 'tex', the prompt strings are rendered using LaTeX.

If the user clicks the **Cancel** button to close an inputdlg box, the dialog returns an empty cell array:

```
answer =
     {}
```

Tips

inputdlg uses the uiwait function to suspend execution until the user responds.

The returned variable answer is a cell array containing strings, one string per text entry field, starting from the top of the dialog box.

To convert a member of the cell array to a number, use str2num. To do this, you can add the following code to the end of any of the examples below:

```
[val status] = str2num(answer{1});  % Use curly bracket for subscript
if ~status
    % Handle empty value returned for unsuccessful conversion
    % ...
end
% val is a scalar or matrix converted from the first input
```

Users can enter scalar or vector values into `inputdlg` fields; `str2num` converts space- and comma-delimited strings into row vectors, and semicolon-delimited strings into column vectors. For example, if `answer{1}` contains `'1 2 3;4 -5 6+7i'`, the conversion produces:

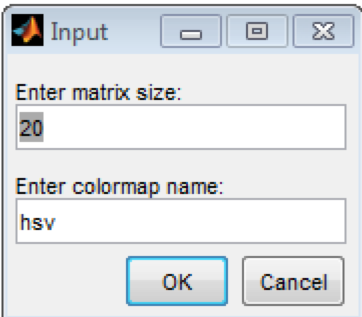
```
val = str2num(answer{1})
val =
   1.0000      2.0000      3.0000
   4.0000     -5.0000      6.0000 + 7.0000i
```

Examples

Example 1

Create a dialog box to input an integer and colormap name. Allow one line for each value.

```
prompt = {'Enter matrix size:','Enter colormap name:'};
dlg_title = 'Input';
num_lines = 1;
def = {'20','hsv'};
answer = inputdlg(prompt,dlg_title,num_lines,def);
```

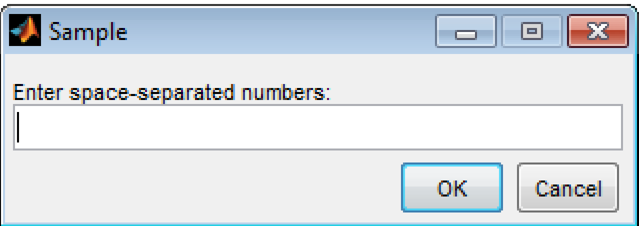


Example 2

Create a dialog box named to accept comma-separated numbers. MATLAB stores accepts the input as a string, so convert the string to numbers using `str2num`.

```
x = inputdlg('Enter space-separated numbers:', 'Sample', [1 50]);
```

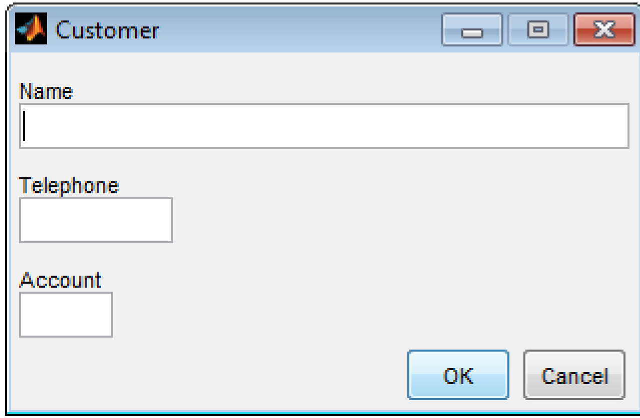
```
data = str2num(x{:});
```



Example 3

Create a dialog box to display input fields of different widths.

```
x = inputdlg({'Name','Telephone','Account'}, 'Customer', [1 50; 1 12;
```

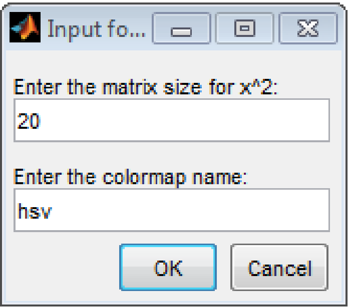


Example 4

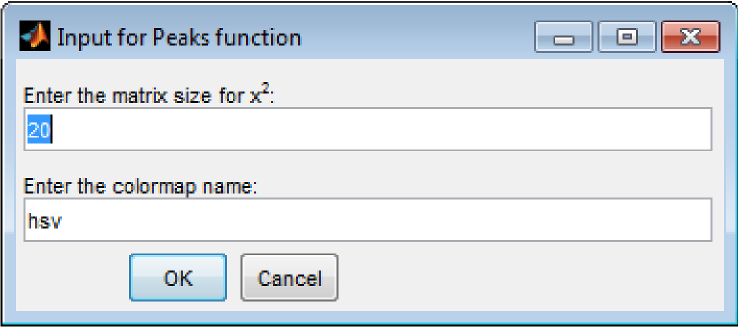
Create a dialog box using the default options. Then, use the options to make it resizable and not modal, and to interpret the text using LaTeX.

```
prompt={'Enter the matrix size for x^2:',...
        'Enter the colormap name:'};
name='Input for Peaks function';
numlines=1;
```

```
defaultanswer={'20','hsv'};
answer=inputdlg(prompt,name,numlines,defaultanswer);
```




```
options.Resize='on';
options.WindowStyle='normal';
options.Interpreter='tex';

answer=inputdlg(prompt,name,numlines,defaultanswer,options);
```



See Also

dialog | errordlg | helpdlg | listdlg | msgbox | questdlg | warndlg | input | figure | str2num | uiwait | uiresume

Purpose

Variable name of function input

Syntax

`inputname(argnum)`

Description

This command can be used only inside the body of a function.

`inputname(argnum)` returns the workspace variable name corresponding to the argument number *argnum*. If the input argument has no name (for example, if it is an expression instead of a variable), the `inputname` command returns the empty string (`''`).

Examples

Suppose the function `myfun.m` is defined as

```
function c = myfun(a,b)
fprintf('First calling variable is "%s"\n.', inputname(1))
```

Then

```
x = 5;  y = 3;  myfun(x,y)
```

produces

```
First calling variable is "x".
```

But

```
myfun(pi+1, pi-1)
```

produces

```
First calling variable is "".
```

See Also

`nargin` | `nargout` | `narginchk`

Purpose

Parse function inputs

Description

Use `inputParser` objects to verify that each input passed to your custom function conforms to the expected type and value.

1. Construct an `inputParser` object.
2. Add inputs to the input scheme using the `addRequired`, `addOptional`, or `addParamValue` methods. Each method allows you to specify a validation function that checks the input. You can call these methods in any order to add inputs to the scheme. However, when you call your custom function, pass any required inputs first. Then, pass any optional positional inputs, followed by parameter value inputs.
3. Optionally, set properties to adjust the parsing behavior, such as handling case sensitivity, structure array inputs, and inputs that are not in the input scheme.
4. Call the `parse` method to parse and store inputs in the `Results`, `UsingDefaults`, and `Unmatched` properties of the `inputParser` object.

Construction

`p = inputParser` creates `inputParser` object `p`.

Properties

CaseSensitive

Logical value that indicates whether to match case when checking argument names.

Possible values:

`false` (0)	Names are not sensitive to case: `'a'` matches `'A'`.
`true` (1)	Names are case sensitive: `'a'` does not match `'A'`.

Default: `false`

FunctionName

String that specifies the name of the function to include in error messages.

Default: If you construct the `inputParser` object within a function, the default value is the name of that function. Otherwise, the default value is an empty string, `''`.

KeepUnmatched

Logical value that indicates how to handle parameter name and value inputs that are not in the input scheme.

Possible values:

`false` (0)	Throw an error whenever inputs are not in the scheme.
`true` (1)	Store the parameter names and values of unmatched inputs in the `Unmatched` property of the `inputParser` object, and suppress the error.

Default: `false`

Parameters

Cell array of strings that contains the names of arguments currently defined in the input scheme. Read-only.

Each method that adds an input argument to the scheme (`addRequired`, `addOptional`, `addParamValue`) updates the `Parameters` property.

Results

Structure containing names and values of inputs that match the function input scheme, populated by the `parse` method. Read-only.

Each field of the `Results` structure corresponds to the name of an input.

StructExpand

Logical value that specifies whether to interpret a structure array as a single input or as a set of parameter name and value pairs.

`true` (1)	Expand structures into separate inputs. Each field name corresponds to an input parameter name.
`false` (0)	Regard a structure array as a single input argument.

Default: `true`

Unmatched

Structure array containing the names and values of inputs that do not match the function input scheme, populated by the `parse` method. Read-only.

If `KeepUnmatched` is `false` (default) or all inputs match the scheme, then `Unmatched` is a 1-by-1 structure with no fields. Otherwise, each field of the structure corresponds to the name of an input that did not match the scheme.

UsingDefaults

Cell array containing the names of inputs not passed explicitly to the function and assigned default values, populated by the `parse` method. Read-only.

Methods

addOptional	Add optional argument to Input Parser scheme
addParamValue	Add parameter name and value argument to Input Parser scheme

addRequired	Add required argument to Input Parser scheme
createCopy	Create copy of `inputParser` object (to be removed)
parse	Parse function inputs

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

The `copy` method creates another scheme with the same properties as an existing `inputParser` object:

```
pNew = copy(pOld);
```

Examples

Input Validation

Check the validity of required and optional function inputs.

Create a custom function with required and optional inputs in the file `findArea.m`.

```
function a = findArea(width,varargin)
   p = inputParser;
   defaultHeight = 1;
   defaultUnits = 'inches';
   defaultShape = 'rectangle';
   expectedShapes = {'square','rectangle','parallelogram'};

   addRequired(p,'width',@isnumeric);
   addOptional(p,'height',defaultHeight,@isnumeric);
   addParamValue(p,'units',defaultUnits);
   addParamValue(p,'shape',defaultShape,...
                 @(x) any(validatestring(x,expectedShapes)));

   parse(p,width,varargin{:});
   a = p.Results.width .* p.Results.height;
```

The input parser checks whether `width` and `height` are numeric, and whether the `shape` matches a string in cell array `expectedShapes`. `@` indicates a function handle, and the syntax `@(x)` creates an anonymous function with input `x`.

Call the function with inputs that do not match the scheme. For example, specify a nonnumeric value for the `width` input:

```
findArea('text')

Error using findArea (line 14)
Argument 'width' failed validation isnumeric.
```

Specify an unsupported value for `shape`:

```
findArea(4,'shape','circle')

Error using findArea (line 14)
Argument 'shape' failed validation with error:
Expected input to match one of these strings:

square, rectangle, parallelogram

The input, 'circle', did not match any of the valid strings.
```

Extra Parameter Value Inputs

Store parameter name and value inputs that are not in the input scheme instead of throwing an error.

```
default = 0;
value = 1;

p = inputParser;
p.KeepUnmatched = true;
addOptional(p,'expectedInputName',default);
parse(p,'extraInput',value);
```

View the unmatched parameter name and value:

```
p.Unmatched

ans = 
    extraInput: 1
```

Case Sensitivity

Enforce case sensitivity when checking function inputs.

```
p = inputParser;
p.CaseSensitive = true;
defaultValue = 0;
addParamValue(p,'InputName',defaultValue);

parse(p,'inputname',10)

Argument 'inputname' did not match any valid parameter of the parser.
```

Structure Array Inputs

Parse structure array inputs with the `StructExpand` property set to `true` (default) or `false`.

Expand a structure array input into parameter name and value pairs using the default `true` value of the `StructExpand` property.

```
s.input1 = 10;
s.input2 = 20;
default = 0;

p = inputParser;
addParamValue(p,'input1',default);
addParamValue(p,'input2',default);
parse(p,s);

p.Results

ans = 
    input1: 10
    input2: 20
```

Explicitly specifying a parameter name and value pair overrides values in the structure.

```
parse(p,s,'input2',300);
p.Results

ans =
    input1: 10
    input2: 300
```

Accept a structure array input as a single argument by setting the `StructExpand` property to `false`.

```
s2.first = 1;
s2.random = rand(3,4,2);
s2.mytext = 'some text';

p = inputParser;
p.StructExpand = false;
addRequired(p,'structInput');
parse(p,s2);

results = p.Results
fieldList = fieldnames(p.Results.structInput)

results =
    structInput: [1x1 struct]

fieldList =
    'first'
    'random'
    'mytext'
```

See Also

validateattributes | validatestring | varargin | narginchk | nargin

Concepts

- "Input Parser Validation Functions"

Purpose

Open Property Inspector

Syntax

```
inspect
inspect(h)
inspect([h1,h2,...])
```

Description

`inspect` creates a separate Property Inspector window to enable the display and modification of the properties of any object you select in the figure window or Layout Editor. If no object is selected, the Property Inspector is blank.

`inspect(h)` creates a Property Inspector window for the object whose handle is `h`.

`inspect([h1,h2,...])` displays properties that objects `h1` and `h2` have in common, or a blank window if there are no such properties; any number of objects can be inspected and edited in this way (for example, handles returned by the `bar` command).

The Property Inspector has the following behaviors:

- Only one Property Inspector window is active at any given time; when you inspect a new object, its properties replace those of the object last inspected.
- When the Property Inspector is open and plot edit mode is on, clicking any object in the figure window displays the properties of that object (or set of objects) in the Property Inspector.
- When you select and inspect two or more objects of different types, the Property Inspector only shows the properties that all objects have in common.
- To change the value of any property, click on the property name shown at the left side of the window, and then enter the new value in the field at the right.

The Property Inspector provides two different views:

- List view — properties are ordered alphabetically (default); this is the only view available for annotation objects.

- Group view — properties are grouped under classified headings (Handle Graphics objects only)

To view alphabetically, click the "AZ" Icon in the Property Inspector toolbar. To see properties in groups, click

the "++" icon. When properties are grouped, the "-" and "+" icons are enabled; click to expand all categories and click to collapse all categories. You can also expand and collapse individual categories by clicking on the "+" next to the category name. Some properties expand and collapse

---

Notes To see a complete description of any property, right-click on its name or value and select **What's This**; a help window opens that displays the reference page entry for it.

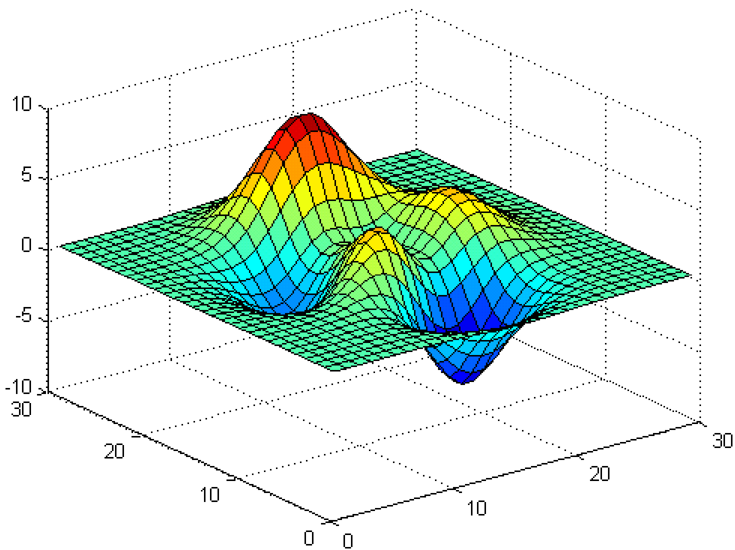
The Property Inspector displays most, but not all, properties of Handle Graphics objects. For example, the `parent` and `children` of HG objects are not shown.
`inspect h` displays a Property Inspector window that enables modification of the string `'h'`, not the object whose handle is `h`.
If you modify properties at the MATLAB command line, you must refresh the Property Inspector window to see the change reflected there. Refresh the Property Inspector by reinvoking `inspect` on the object.

Examples

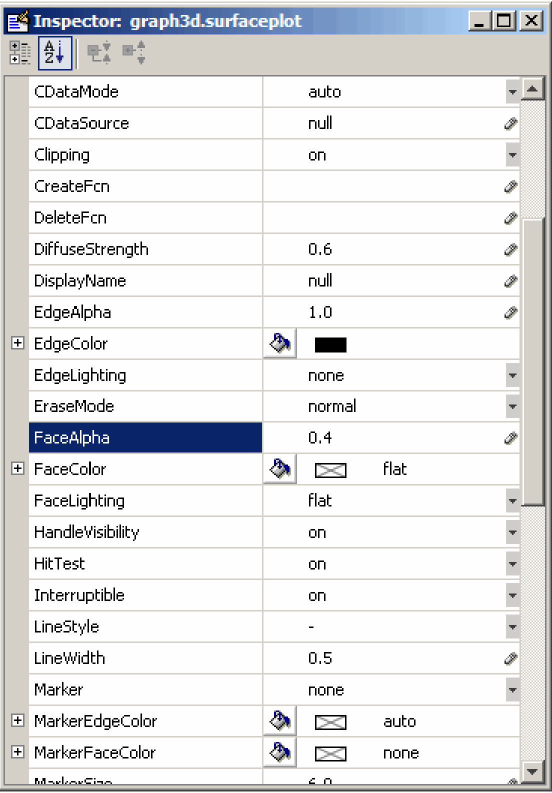
Example 1

Create a surface mesh plot and view its properties with the Property Inspector:

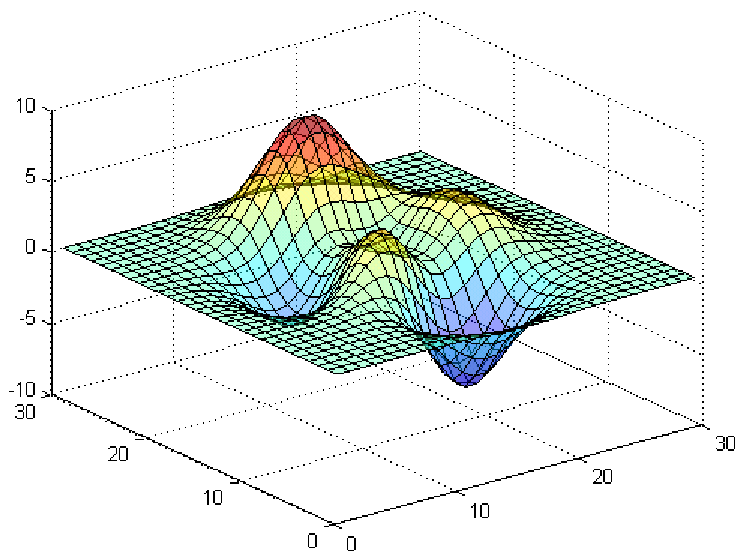
```
Z = peaks(30);
h = surf(Z)
inspect(h)
```



Use the Property Inspector to change the `FaceAlpha` property from `1.0` to `0.4` (equivalent to the command `set(h,'FaceAlpha',0.4)`). `FaceAlpha` controls the transparency of patch faces.



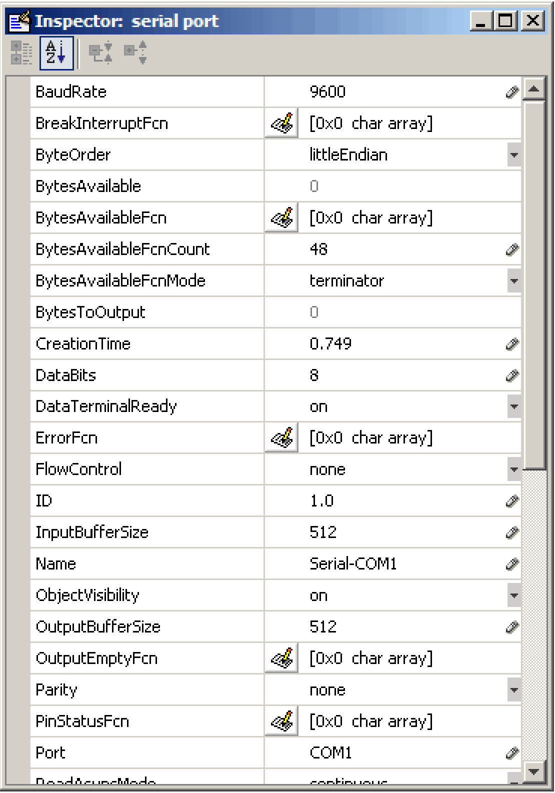
When you press **Enter** or click a different field, the `FaceAlpha` property of the surface object is updated:



Example 2

Create a serial port object for `COM1` on a Windows platform and use the Property Inspector to peruse its properties:

```
s = serial('COM1');
inspect(s)
```



Because COM objects do not define property groupings, only the alphabetical list view of their properties is available.

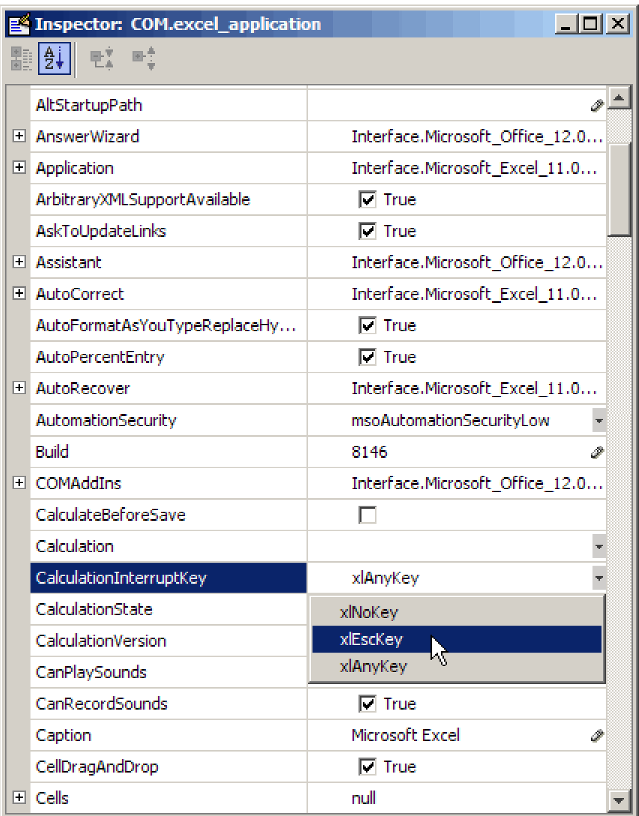
Example 3

Create a COM Excel server and open a Property Inspector window with `inspect`:

```
h = actxserver('excel.application');
inspect(h)
```

Scroll down until you see the `CalculationInterruptKey` property, which by default is `xlAnyKey`. Click on the down-arrow in the right

margin of the property inspector and select `xlEscKey` from the drop-down menu, as shown below:



Check this field in the MATLAB Command Window using `get` to confirm that it has changed:

```
get(h,'CalculationInterruptKey')

ans =
xlEscKey
```

See Also

`get` | `set` | `isprop` | `guide` | `addproperty` | `deleteproperty`

Purpose

Event information when event occurs

Syntax

```
instrcallback(obj,event)
```

Description

`instrcallback(obj,event)` displays a message that contains the event type, `event`, the time the event occurred, and the name of the serial port object, `obj`, that caused the event to occur.

For error events, the error message is also displayed. For pin status events, the pin that changed value and its value are also displayed.

Tips

You should use `instrcallback` as a template from which you create callback functions that suit your specific application needs.

Examples

The following example creates the serial port objects `s`, on a Windows platform. It configures `s` to execute `instrcallback` when an output-empty event occurs. The event occurs after the `*IDN?` command is written to the instrument.

```
s = serial('COM1');
set(s,'OutputEmptyFcn',@instrcallback)
fopen(s)
fprintf(s,'*IDN?','async')
```

The resulting display from `instrcallback` is shown below.

```
OutputEmpty event occurred at 08:37:49 for the object:
Serial-COM1.
```

Read the identification information from the input buffer and end the serial port session.

```
idn = fscanf(s);
fclose(s)
delete(s)
clear s
```

instrfind

Purpose

Read serial port objects from memory to MATLAB workspace

Syntax

```
out = instrfind
out = instrfind('PropertyName',PropertyValue,...)
out = instrfind(S)
out = instrfind(obj,'PropertyName',PropertyValue,...)
```

Description

`out = instrfind` returns all valid serial port objects as an array to `out`.

`out = instrfind('`*`PropertyName`*`',PropertyValue,...)` returns an array of serial port objects whose property names and property values match those specified.

`out = instrfind(S)` returns an array of serial port objects whose property names and property values match those defined in the structure `S`. The field names of `S` are the property names, while the field values are the associated property values.

`out = instrfind(obj,'`*`PropertyName`*`',PropertyValue,...)` restricts the search for matching property name/property value pairs to the serial port objects listed in `obj`.

Tips

Refer to "Displaying Property Names and Property Values" for a list of serial port object properties that you can use with `instrfind`.

You must specify property values using the same format as the `get` function returns. For example, if `get` returns the `Name` property value as `MyObject`, `instrfind` will not find an object with a `Name` property value of `myobject`. However, this is not the case for properties that have a finite set of string values. For example, `instrfind` will find an object with a `Parity` property value of `Even` or `even`.

You can use property name/property value string pairs, structures, and cell array pairs in the same call to `instrfind`.

Examples

Suppose you create the following two serial port objects on a Windows platform.

```
s1 = serial('COM1');
```

```
s2 = serial('COM2');
set(s2,'BaudRate',4800)
fopen([s1 s2])
```

You can use `instrfind` to return serial port objects based on property values.

```
out1 = instrfind('Port','COM1');
out2 = instrfind({'Port','BaudRate'},{'COM2',4800});
```

You can also use `instrfind` to return cleared serial port objects to the MATLAB workspace.

```
clear s1 s2
newobjs = instrfind

   Instrument Object Array
   Index:   Type:          Status:     Name:
   1        serial         open        Serial-COM1
   2        serial         open        Serial-COM2
```

To close both `s1` and `s2`

```
fclose(newobjs)
```

See Also

`clear` | `get`

instrfindall

Purpose

Find visible and hidden serial port objects

Syntax

```
out = instrfindall
out = instrfindall('P1',V1,...)
out = instrfindall(s)
out = instrfindall(objs,'P1',V1,...)
```

Description

`out = instrfindall` finds all serial port objects, regardless of the value of the objects' `ObjectVisibility` property. The object or objects are returned to `out`.

`out = instrfindall('P1',V1,...)` returns an array, `out`, of serial port objects whose property names and corresponding property values match those specified as arguments.

`out = instrfindall(s)` returns an array, `out`, of serial port objects whose property names and corresponding property values match those specified in the structure `s`, where the field names correspond to property names and the field values correspond to the current value of the respective property.

`out = instrfindall(objs,'P1',V1,...)` restricts the search for objects with matching property name/value pairs to the serial port objects listed in `objs`.

Note that you can use string property name/property value pairs, structures, and cell array property name/property value pairs in the same call to `instrfindall`.

Tips

`instrfindall` differs from `instrfind` in that it finds objects whose `ObjectVisibility` property is set to `off`.

Property values are case sensitive. You must specify property values using the same format as that returned by the `get` function. For example, if `get` returns the `Name` property value as `'MyObject'`, `instrfindall` will not find an object with a `Name` property value of `'myobject'`. However, this is not the case for properties that have a finite set of string values. For example, `instrfindall` will find an object with a `Parity` property value of `'Even'` or `'even'`.

Examples

Suppose you create the following serial port objects on a Windows platform:

```
s1 = serial('COM1');
s2 = serial('COM2');
set(s2,'ObjectVisibility','off')
```

Because object `s2` has its `ObjectVisibility` set to `'off'`, it is not visible to commands like `instrfind`:

```
instrfind

   Serial Port Object : Serial-COM1
```

However, `instrfindall` finds all objects regardless of the value of `ObjectVisibility`:

```
instrfindall

 Instrument Object Array
 Index:   Type:          Status:    Name:
 1        serial         closed     Serial-COM1
 2        serial         closed     Serial-COM2
```

The following statements use `instrfindall` to return objects with specific property settings, which are passed as cell arrays:

```
props = {'PrimaryAddress','SecondaryAddress};
vals = {2,0};
obj = instrfindall(props,vals);
```

You can use `instrfindall` as an argument when you want to apply the command to all objects, visible and invisible. For example, the following statement makes all objects visible:

```
set(instrfindall,'ObjectVisibility','on')
```

See Also

`get` | `instrfind` | `ObjectVisibility`

int2str

Purpose

Convert integer to string

Syntax

```
str = int2str(N)
```

Description

`str = int2str(N)` converts an integer to a string with integer format. The input `N` can be a single integer or a vector or matrix of integers. Noninteger inputs are rounded before conversion.

Examples

`int2str(2+3)` is the string `'5'`.

One way to label a plot is

```
title(['case number ' int2str(n)])
```

For matrix or vector inputs, `int2str` returns a string matrix:

```
int2str(eye(3))

ans =

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

See Also

`fprintf` | `num2str` | `sprintf` | `cast`

Purpose

Convert to 8-bit signed integer

Syntax

```
intArray = int8(array)
```

Description

`intArray = int8(array)` converts the elements of an array into signed 8-bit (1-byte) integers of class `int8`.

Input Arguments

array

Array of any numeric class, such as `single` or `double`. If `array` is already of class `int8`, the `int8` function has no effect.

Output Arguments

intArray

Array of class `int8`. Values range from -2^7 to $2^7 - 1$.

The `int8` function maps any values in `array` that are outside the limit to the nearest endpoint. For example,

```
int8(2^7)   % 2^7 = 128
```

returns

```
ans =
    127
```

Examples

Convert a `double` array to `int8`:

```
mydata = int8(magic(10));
```

Alternatives

When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as `zeros`, `ones` or `eye`), rather than calling an integer conversion function. For example,

```
I = int8(zeros(100));     % Creates an intermediate array
```

is not as efficient as

```
I = zeros(100, 'int8');   % Preferred
```

See Also

`double` | `single` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `intmax` | `intmin`

Tutorials

- "Integers"
- "Arithmetic Operations on Integer Classes"

Purpose

Convert to 16-bit signed integer

Syntax

```
intArray = int16(array)
```

Description

`intArray = int16(array)` converts the elements of an array into signed 16-bit (2-byte) integers of class `int16`.

Input Arguments

array

Array of any numeric class, such as `single` or `double`. If `array` is already of class `int16`, the `int16` function has no effect.

Output Arguments

intArray

Array of class `int16`. Values range from -2^{15} to $2^{15} - 1$.

The `int16` function maps any values in `array` that are outside the limit to the nearest endpoint. For example,

```
int16(2^15)   % 2^15 = 32768
```

returns

```
ans =
    32767
```

Examples

Convert a `double` array to `int16`:

```
mydata = int16(magic(100));
```

Alternatives

When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as `zeros`, `ones` or `eye`), rather than calling an integer conversion function. For example,

```
I = int16(zeros(100));     % Creates an intermediate array
```

is not as efficient as

```
I = zeros(100, 'int16');   % Preferred
```

See Also

`double` | `single` | `int8` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `intmax` | `intmin`

Tutorials

- "Integers"
- "Arithmetic Operations on Integer Classes"

int32

Purpose

Convert to 32-bit signed integer

Syntax

```
intArray = int32(array)
```

Description

`intArray = int32(array)` converts the elements of an array into signed 32-bit (4-byte) integers of class `int32`.

Input Arguments

array

Array of any numeric class, such as `single` or `double`. If `array` is already of class `int32`, the `int32` function has no effect.

Output Arguments

intArray

Array of class `int32`. Values range from -2^{31} to $2^{31} - 1$.

The `int32` function maps any values in `array` that are outside the limit to the nearest endpoint. For example,

```
int32(2^31)   % 2^31 = 2147483648
```

returns

```
ans =
    2147483647
```

Examples

Convert a `double` array to `int32`:

```
mydata = int32(magic(1000));
```

Alternatives

When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as `zeros`, `ones` or `eye`), rather than calling an integer conversion function. For example,

```
I = int32(zeros(100));     % Creates an intermediate array
```

is not as efficient as

```
I = zeros(100, 'int32');   % Preferred
```

See Also

`double` | `single` | `int8` | `int16` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `intmax` | `intmin`

Tutorials

- "Integers"
- "Arithmetic Operations on Integer Classes"

Purpose

Convert to 64-bit signed integer

Syntax

```
intArray = int64(array)
```

Description

`intArray = int64(array)` converts the elements of an array into signed 64-bit (8-byte) integers of class `int64`.

Tips

Double-precision floating-point numbers have only 52 bits in the mantissa. Therefore, `double` values cannot represent all integers greater than 2^{53} correctly. Before performing arithmetic operations on values larger than 2^{53} in magnitude, convert the values to 64-bit integers. For example,

```
x = int64(2^53+1);     % Floating-point arithmetic, loses precision
```

is not as accurate as the 64-bit integer arithmetic operation:

```
x = int64(2^53) + 1;   % Preferred
```

Input Arguments

array

Array of any numeric class, such as `single` or `double`. If `array` is already of class `int64`, the `int64` function has no effect.

Output Arguments

intArray

Array of class `int64`. Values range from -2^{63} to $2^{63} - 1$.

The `int64` function maps any values in `array` that are outside the limit to the nearest endpoint. For example,

```
int64(2^63)   % 2^63 = 9223372036854775808
```

returns

```
ans =
    9223372036854775807
```

Examples

Convert a literal value to int64:

```
x = int64(9007199254740993);
```

Alternatives

When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as zeros, ones or eye), rather than calling an integer conversion function. For example,

```
I = int64(zeros(100));     % Creates an intermediate array
```

is not as efficient as

```
I = zeros(100, 'int64');   % Preferred
```

See Also

double | single | int8 | int16 | int32 | uint8 | uint16 | uint32 | uint64 | intmax | intmin

Tutorials

- “Integers”
- “Arithmetic Operations on Integer Classes”

Purpose

Numerically evaluate integral

Syntax

```
q = integral(fun,xmin,xmax)
q = integral(fun,xmin,xmax,Name,Value)
```

Description

`q = integral(fun,xmin,xmax)` approximates the integral of function `fun` from `xmin` to `xmax` using global adaptive quadrature and default error tolerances.

`q = integral(fun,xmin,xmax,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments.

Input Arguments

fun - Integrand

function handle

Integrand, specified as a function handle, defines the function to be integrated from `xmin` to `xmax`. For scalar-valued problems, the function `y = fun(x)` must accept a vector argument, `x`, and return a vector result, `y`. If you set the `'ArrayValued'` option to `true`, `fun` must accept a scalar and return an array of fixed size.

xmin - Lower limit of *x*

real number | complex number

Lower limit of *x*, specified as a real (finite or infinite) scalar value or a complex (finite) scalar value. If either `xmin` or `xmax` are complex, `integral` approximates the path integral from `xmin` to `xmax` over a straight line path.

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

xmax - Upper limit of *x*

real number | complex number

Upper limit of *x*, specified as a real (finite or infinite) scalar value or a complex (finite) scalar value. If either `xmin` or `xmax` are complex,

`integral` approximates the path integral from `xmin` to `xmax` over a straight line path.

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

Name-Value Pair Arguments

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

Example: `'AbsTol',1e-12` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

'AbsTol' - Absolute error tolerance

nonnegative real number

Absolute error tolerance, specified as the comma-separated pair consisting of `'AbsTol'` and a nonnegative real number. `integral` uses the absolute error tolerance to limit an estimate of the absolute error, $|q - Q|$, where q is the computed value of the integral and Q is the (unknown) exact value. `integral` might provide more decimal places of precision if you decrease the absolute error tolerance. The default value is `1e-10`.

Note `AbsTol` and `RelTol` work together. `integral` might satisfy the absolute error tolerance or the relative error tolerance, but not necessarily both. For more information on using these tolerances, see the "Tips" on page 1-2748 section.

Example: `'AbsTol',1e-12` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

Data Types
`single | double`

'RelTol' - Relative error tolerance

nonnegative real number

Relative error tolerance, specified as the comma-separated pair consisting of `'RelTol'` and a nonnegative real number. `integral` uses the relative error tolerance to limit an estimate of the relative error, $|q - Q|/|Q|$, where q is the computed value of the integral and Q is the (unknown) exact value. `integral` might provide more significant digits of precision if you decrease the relative error tolerance. The default value is `1e-6`.

Note `RelTol` and `AbsTol` work together. `integral` might satisfy the relative error tolerance or the absolute error tolerance, but not necessarily both. For more information on using these tolerances, see the Tips section.

Example: `'RelTol',1e-9` sets the relative error tolerance to approximately 9 significant digits.

Data Types
`single | double`

'ArrayValued' - Array-valued function flag

`false` (default) | `true`

Array-valued function flag, specified as the comma-separated pair consisting of `'ArrayValued'` and either `true` or `false`. Set this flag to `true` when you want to integrate over an array-valued function. The shape of `fun(x)` can be a vector, matrix, or N-D array.

Example: `'ArrayValued',true` indicates that the integrand is an array-valued function.

Data Types
`logical`

'Waypoints' - Integration waypoints

vector

Integration waypoints, specified as the comma-separated pair consisting of `'Waypoints'` and a vector of real or complex numbers. Use waypoints to indicate any points in the integration interval you would like the integrator to use. You can use waypoints to integrate efficiently across discontinuities of the integrand. Specify the locations of the discontinuities in the vector you supply.

You can specify waypoints when you want to perform complex contour integration. If `xmin`, `xmax`, or any entry of the waypoints vector is complex, the integration is performed over a sequence of straight line paths in the complex plane.

Example: `'Waypoints',[1+1i,1-1i]` specifies two complex waypoints along the interval of integration.

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

Output Arguments

q - Computed integral

Computed integral of `fun(x)` between the limits `xmin` and `xmax`, returned as a numeric value or numeric array.

Tips

- Do not use waypoints to specify singularities. Instead, split the interval and add the results of separate integrations with the singularities at the endpoints.
- The `integral` function attempts to satisfy:

  ```
  abs(q - Q) <= max(AbsTol,RelTol*abs(q))
  ```

  where `q` is the computed value of the integral and `Q` is the (unknown) exact value. The absolute and relative tolerances provide a way of

trading off accuracy and computation time. Usually, the relative tolerance determines the accuracy of the integration. However if `abs(q)` is sufficiently small, the absolute tolerance determines the accuracy of the integration. You should generally specify both absolute and relative tolerances together.

- If you specify a complex value for `xmin`, `xmax`, or any waypoint, all of your limits and waypoints must be finite.
- If you are specifying single-precision limits of integration, or if `fun` returns single-precision results, you might need to specify larger absolute and relative error tolerances.

Examples

Evaluate Improper Integral

Create the anonymous function $f(x) = e^{-x^2}(\ln x)^2$.

```
fun = @(x) exp(-x.^2).*log(x).^2;
```

Evaluate the integral from `x=0` to `x=Inf`.

```
q = integral(fun,0,Inf)

q =

    1.9475
```

Integrate Parameterized Function

Create the anonymous function $f(x) = 1/(x^3 - 2x - c)$ with one parameter, c.

```
fun = @(x,c) 1./(x.^3-2*x-c);
```

Evaluate the integral from `x=0` to `x=2` at `c=5`.

```
q = integral(@(x)fun(x,5),0,2)

q =
```

```
   -0.4605
```

Evaluate Integral with Singularity at the Lower Limit of Integration

Create the anonymous function $f(x) = \ln(x)$.

```
fun = @(x)log(x);
```

Evaluate the integral from `x=0` to `x=1` with the default error tolerances.

```
format long
q1 = integral(fun,0,1)
```

```
q1 =

  -1.000000010959678
```

Evaluate the integral again, specifying approximately 12 decimal places of accuracy.

```
q2 = integral(fun,0,1,'RelTol',0,'AbsTol',1e-12)
```

```
q2 =

  -1.000000000000010
```

Complex Contour Integration Using Waypoints

Specify waypoints `[1+1i,1-1i]` to integrate over the triangular path: `0` to `1+1i` to `1-1i` to `0`.

Create the anonymous function $f(z) = 1/(2z - 1)$.

```
fun = @(z) 1./(2*z-1);
```

Integrate in the complex plane over the triangular path from `0` to `1+1i` to `1-1i` to `0`.

```
q = integral(fun,0,0,'Waypoints',[1+1i,1-1i])
```

```
q =

        0 - 3.1416i
```

Integrate Vector-Valued Function

Specify `'ArrayValued',true` to evaluate the integral of an array-valued or vector-valued function.

Create the anonymous vector-valued function $f(x) = [\sin x, \sin 2x, \sin 3x, \sin 4x, \sin 5x]$ and integrate from `x=0` to `x=1`.

```
fun = @(x)sin((1:5)*x);
q = integral(fun,0,1,'ArrayValued',true)
```

```
q =

    0.4597    0.7081    0.6633    0.4134    0.1433
```

Improper Integral of an Oscillatory Function

Create the anonymous function $f(x) = x^5 e^{-x} \sin x$.

```
fun = @(x)x.^5.*exp(-x).*sin(x);
```

Evaluate the integral from `x=0` to `x=Inf` , adjusting the absolute and relative tolerances.

```
format long
q = integral(fun,0,Inf,'RelTol',1e-8,'AbsTol',1e-13)
```

```
q =

  -14.999999999998364
```

See Also

`integral2` | `integral3` | `function_handle` | `trapz`

Concepts

- “Parameterizing Functions”

integral2

Purpose

Numerically evaluate double integral

Syntax

```
q = integral2(fun,xmin,xmax,ymin,ymax)
q = integral2(fun,xmin,xmax,ymin,ymax,Name,Value)
```

Description

`q = integral2(fun,xmin,xmax,ymin,ymax)` approximates the integral of the function `z = fun(x,y)` over the planar region `xmin` ≤ `x` ≤ `xmax` and `ymin(x)` ≤ `y` ≤ `ymax(x)`.

`q = integral2(fun,xmin,xmax,ymin,ymax,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments.

Input Arguments

fun - Integrand

function handle

Integrand, specified as a function handle, defines the function to be integrated over the planar region `xmin` ≤ `x` ≤ `xmax` and `ymin(x)` ≤ `y` ≤ `ymax(x)`. The function `fun` must accept two arrays of the same size and return an array of corresponding values. It must perform element-wise operations.

Data Types

`function_handle`

xmin - Lower limit of *x*

real number

Lower limit of *x*, specified as a real scalar value that is either finite or infinite.

Data Types

`double | single`

xmax - Upper limit of *x*

real number

Upper limit of *x*, specified as a real scalar value that is either finite or infinite.

Data Types
`double | single`

ymin - Lower limit of *y*

real number | function handle

Lower limit of y, specified as a real scalar value that is either finite or infinite. You can specify `ymin` to be a function handle (a function of x) when integrating over a nonrectangular region.

Data Types
`double | function_handle | single`

ymax - Upper limit of *y*

real number | function handle

Upper limit of y, specified as a real scalar value that is either finite or infinite. You also can specify `ymax` to be a function handle (a function of x) when integrating over a nonrectangular region.

Data Types
`double | function_handle | single`

Name-Value Pair Arguments

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

Example: `'AbsTol',1e-12` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

'AbsTol' - Absolute error tolerance

nonnegative real number

Absolute error tolerance, specified as the comma-separated pair consisting of `'AbsTol'` and a nonnegative real number. `integral2` uses the absolute error tolerance to limit an estimate of the absolute error, $|q - Q|$, where q is the computed value of the integral and Q is

the (unknown) exact value. `integral2` might provide more decimal places of precision if you decrease the absolute error tolerance. The default value is `1e-10`.

Note `AbsTol` and `RelTol` work together. `integral2` might satisfy the absolute error tolerance or the relative error tolerance, but not necessarily both. For more information on using these tolerances, see the "Tips" on page 1-2756 section.

Example: `'AbsTol',1e-12` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

Data Types
`double | single`

'RelTol' - Relative error tolerance
nonnegative real number

Relative error tolerance, specified as the comma-separated pair consisting of `'RelTol'` and a nonnegative real number. `integral2` uses the relative error tolerance to limit an estimate of the relative error, $|q - Q|/|Q|$, where q is the computed value of the integral and Q is the (unknown) exact value. `integral2` might provide more significant digits of precision if you decrease the relative error tolerance. The default value is `1e-6`.

Note `RelTol` and `AbsTol` work together. `integral2` might satisfy the relative error tolerance or the absolute error tolerance, but not necessarily both. For more information on using these tolerances, see the "Tips" on page 1-2756 section.

Example: `'RelTol',1e-9` sets the relative error tolerance to approximately 9 significant digits.

Data Types
double | single

'Method' - Integration method

'auto' (default) | 'tiled' | 'iterated'

Integration method, specified as the comma-separated pair consisting of 'Method' and one of the methods described below.

Integration Method	Description
'auto'	For most cases, integral2 uses the 'tiled' method. It uses the 'iterated' method when any of the integration limits are infinite. This is the default method.
'tiled'	integral2 transforms the region of integration to a rectangular shape and subdivides it into smaller rectangular regions as needed. The integration limits must be finite.
'iterated'	integral2 calls integral to perform an iterated integral. The outer integral is evaluated over xmin ≤ x ≤ xmax. The inner integral is evaluated over ymin(x) ≤ y ≤ ymax(x). The integration limits can be infinite.

Example: 'Method','tiled' specifies the tiled integration method.

Output Arguments

q - Computed integral

Computed integral of fun(x,y) over the specified region, returned as a numeric value.

Tips

- The `integral2` function attempts to satisfy:

  ```
  abs(q - Q) <= max(AbsTol,RelTol*abs(q))
  ```

  where `q` is the computed value of the integral and `Q` is the (unknown) exact value. The absolute and relative tolerances provide a way of trading off accuracy and computation time. Usually, the relative tolerance determines the accuracy of the integration. However if `abs(q)` is sufficiently small, the absolute tolerance determines the accuracy of the integration. You should generally specify both absolute and relative tolerances together.

- The `'iterated'` method can be more effective when your function has discontinuities within the integration region. However, the best performance and accuracy occurs when you split the integral at the points of discontinuity and sum the results of multiple integrations.

- When integrating over nonrectangular regions, the best performance and accuracy occurs when `ymin`, `ymax`, (or both) are function handles. Avoid setting integrand function values to zero to integrate over a nonrectangular region. If you must do this, specify `'iterated'` method.

- Use the `'iterated'` method when `ymin`, `ymax`, (or both) are unbounded functions.

- When paramaterizing anonymous functions, be aware that parameter values persist for the life of the function handle. For example, the function `fun = @(x,y) x + y + a` uses the value of `a` at the time `fun` was created. If you later decide to change the value of `a`, you must redefine the anonymous function with the new value.

- If you are specifying single-precision limits of integration, or if `fun` returns single-precision results, you might need to specify larger absolute and relative error tolerances.

Examples

Integrate Triangular Region with Singularity at the Boundary

The function

$$f(x,y) = \frac{1}{(\sqrt{x+y})\ (1+x+y)}$$

is undefined when x and y are zero. `integral2` performs best when singularities are on the integration boundary.

Create the anonymous function.

```
fun = @(x,y) 1./( sqrt(x + y) .* (1 + x + y).^2 )
```

Integrate over the triangular region bounded by $0 \le x \le 1$ and $0 \le y \le 1 - x$.

```
ymax = @(x) 1 - x
q = integral2(fun,0,1,0,ymax)
```

```
q =

    0.2854
```

Evaluate Double Integral in Polar Coordinates

Define the function

$$f(\theta,r) = \frac{r}{\sqrt{r\cos\theta + r\sin\theta}\ \ (1 + r\cos\theta + r\sin\theta)^2}$$

```
fun = @(x,y) 1./( sqrt(x + y) .* (1 + x + y).^2 );
polarfun = @(theta,r) fun(r.*cos(theta),r.*sin(theta)).*r;
```

Define a function for the upper limit of r.

```
rmax = @(theta) 1./(sin(theta) + cos(theta));
```

Integrate over the region bounded by $0 \le \theta \le \pi/2$ and $0 \le r \le \text{rmax}$.

```
q = integral2(polarfun,0,pi/2,0,rmax)
```

```
q =
```

```
    0.2854
```

Evaluate Double Integral of Parameterized Function with Specific Method and Error Tolerance

Create the anonymous parameterized function $f(x,y) = ax^2 + by^2$ with parameters a=3 and b=5.

```
a = 3; b = 5;
fun = @(x,y) a*x.^2 + b*y.^2;
```

Evaluate the integral over the region $0 \le x \le 5$ and $-5 \le y \le 0$. Specify the `'iterated'` method and approximately 10 significant digits of accuracy.

```
format long
q = integral2(fun,0,5,-5,0,'Method','iterated',...
'AbsTol',0,'RelTol',1e-10)
```

```
q =

     1.666666666666666e+03
```

See Also

`integral` | `integral3` | `function_handle` | `trapz`

Concepts

- “Parameterizing Functions”

Purpose

Numerically evaluate triple integral

Syntax

```
q = integral3(fun,xmin,xmax,ymin,ymax,zmin,zmax)
q = integral3(fun,xmin,xmax,ymin,ymax,zmin,zmax,Name,Value)
```

Description

`q = integral3(fun,xmin,xmax,ymin,ymax,zmin,zmax)` approximates the integral of the function `z = fun(x,y,z)` over the region `xmin` ≤ `x` ≤ `xmax`, `ymin(x)` ≤ `y` ≤ `ymax(x)` and `zmin(x,y)` ≤ `z` ≤ `zmax(x,y)`.

`q = integral3(fun,xmin,xmax,ymin,ymax,zmin,zmax,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments.

Input Arguments

fun - Integrand

function handle

Integrand, specified as a function handle, defines the function to be integrated over the region `xmin` ≤ `x` ≤ `xmax`, `ymin(x)` ≤ `y` ≤ `ymax(x)`, and `zmin(x,y)` ≤ `z` ≤ `zmax(x,y)`. The function `fun` must accept three arrays of the same size and return an array of corresponding values. It must perform element-wise operations.

Data Types

`function_handle`

xmin - Lower limit of *x*

real number

Lower limit of *x*, specified as a real scalar value that is either finite or infinite.

Example:

Data Types

`double | single`

xmax - Upper limit of *x*

real number

Upper limit of *x*, specified as a real scalar value that is either finite or infinite.

Data Types
`double | single`

ymin - Lower limit of *y*

real number | function handle

Lower limit of *y*, specified as a real scalar value that is either finite or infinite. You also can specify `ymin` to be a function handle (a function of *x*) when integrating over a nonrectangular region.

Example:

Data Types
`double | function_handle | single`

ymax - Upper limit of *y*

real number | function handle

Upper limit of *y*, specified as a real scalar value that is either finite or infinite. You also can specify `ymax` to be a function handle (a function of *x*) when integrating over a nonrectangular region.

Example:

Data Types
`double | function_handle | single`

zmin - Lower limit of *z*

real number | function handle

Lower limit of *z*, specified as a real scalar value that is either finite or infinite. You also can specify `zmin` to be a function handle (a function of *x*,*y*) when integrating over a nonrectangular region.

Data Types
`double | function_handle | single`

zmax - Upper limit of *z*

real number | function handle

Upper limit of *z*, specified as a real scalar value that is either finite or infinite. You also can specify `zmax` to be a function handle (a function of *x*,*y*) when integrating over a nonrectangular region.

Data Types
`double | function_handle | single`

Name-Value Pair Arguments

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

Example: `'AbsTol',1e-12` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

'AbsTol' - Absolute error tolerance

nonnegative real number

Absolute error tolerance, specified as the comma-separated pair consisting of `'AbsTol'` and a nonnegative real number. `integral3` uses the absolute error tolerance to limit an estimate of the absolute error, $|q - Q|$, where q is the computed value of the integral and Q is the (unknown) exact value. `integral3` might provide more decimal places of precision if you decrease the absolute error tolerance. The default value is `1e-10`.

Note `AbsTol` and `RelTol` work together. `integral3` might satisfy the absolute error tolerance or the relative error tolerance, but not necessarily both. For more information on using these tolerances, see the "Tips" on page 1-2763 section.

Example: `'AbsTol',1e-12` sets the absolute error tolerance to approximately 12 decimal places of accuracy.

Data Types
`double | single`

'RelTol' - Relative error tolerance

nonnegative real number

Relative error tolerance, specified as the comma-separated pair consisting of `'RelTol'` and a nonnegative real number. `integral3` uses the relative error tolerance to limit an estimate of the relative error, $|q - Q|/|Q|$, where q is the computed value of the integral and Q is the (unknown) exact value. `integral3` might provide more significant digits of precision if you decrease the relative error tolerance. The default value is `1e-6`.

Note `RelTol` and `AbsTol` work together. `integral3` might satisfy the relative error tolerance or the absolute error tolerance, but not necessarily both. For more information on using these tolerances, see the "Tips" on page 1-2763 section.

Example: `'RelTol',1e-9` sets the relative error tolerance to approximately 9 significant digits.

Data Types
`double | single`

'Method' - Integration method

`'auto'` (default) | `'tiled'` | `'iterated'`

Integration method, specified as the comma-separated pair consisting of `'Method'` and one of the methods described below.

Integration Method	Description
'auto'	For most cases, integral3 uses the 'tiled' method. It uses the 'iterated' method when any of the integration limits are infinite. This is the default method.
'tiled'	integral3 calls integral to integrate over xmin ≤ x ≤ xmax. It calls integral2 with the 'tiled' method to evaluate the double integral over ymin(x) ≤ y ≤ ymax(x) and zmin(x,y) ≤ z ≤ zmax(x,y).
'iterated'	integral3 calls integral to integrate over xmin ≤ x ≤ xmax. It calls integral2 with the 'iterated' method to evaluate the double integral over ymin(x) ≤ y ≤ ymax(x) and zmin(x,y) ≤ z ≤ zmax(x,y). The integration limits can be infinite.

Example: 'Method','tiled' specifies the tiled integration method.

Output Arguments

q - Computed integral

Computed integral of fun(x,y,z) over the specified region, returned as a numeric value.

Tips

- The integral3 function attempts to satisfy:

  ```
  abs(q - Q) <= max(AbsTol,RelTol*abs(q))
  ```

  where q is the computed value of the integral and Q is the (unknown) exact value. The absolute and relative tolerances provide a way of trading off accuracy and computation time. Usually, the relative tolerance determines the accuracy of the integration. However if abs(q) is sufficiently small, the absolute tolerance determines the accuracy of the integration. You should generally specify both absolute and relative tolerances together.

- The 'iterated' method can be more effective when your function has discontinuities within the integration region. However, the best performance and accuracy occurs when you split the integral at the points of discontinuity and sum the results of multiple integrations.
- When integrating over nonrectangular regions, the best performance and accuracy occurs when any or all of the limits: ymin, ymax, zmin, zmax are function handles. Avoid setting integrand function values to zero to integrate over a nonrectangular region. If you must do this, specify 'iterated' method.
- Use the 'iterated' method when any or all of the limits: ymin(x), ymax(x), zmin(x,y), zmax(x,y) are unbounded functions.
- When paramaterizing anonymous functions, be aware that parameter values persist for the life of the function handle. For example, the function fun = @(x,y,z) x + y + z + a uses the value of a at the time fun was created. If you later decide to change the value of a, you must redefine the anonymous function with the new value.
- If you are specifying single-precision limits of integration, or if fun returns single-precision results, you may need to specify larger absolute and relative error tolerances.

Examples

Triple Integral with Finite Limits

Define the anonymous function $f(x,y,z) = y \sin x + z \cos x$.

```
fun = @(x,y,z) y.*sin(x)+z.*cos(x)
```

Integrate over the region $0 \le x \le \pi$, $0 \le y \le 1$, and $-1 \le z \le 1$.

```
q = integral3(fun,0,pi,0,1,-1,1)
```

```
q =

    2.0000
```

Integral Over the Unit Sphere in Cartesian Coordinates

Define the anonymous function $f(x,y,z) = x \cos y + x^2 \cos z$.

```
fun = @(x,y,z) x.*cos(y) + x.^2.*cos(z)
```

Define the limits of integration.

```
xmin = -1;
xmax = 1;
ymin = @(x)-sqrt(1 - x.^2);
ymax = @(x) sqrt(1 - x.^2);
zmin = @(x,y)-sqrt(1 - x.^2 - y.^2);
zmax = @(x,y) sqrt(1 - x.^2 - y.^2);
```

Evaluate the definite integral with the `'tiled'` method.

```
q = integral3(fun,xmin,xmax,ymin,ymax,zmin,zmax,'Method','tiled')
```

```
q =

    0.7796
```

Evaluate Improper Triple Integral of Parameterized Function

Define the anonymous parameterized function $f(x,y,z) = 10/(x^2 + y^2 + z^2 + a)$.

```
a = 2;
f = @(x,y,z) 10./(x.^2 + y.^2 + z.^2 + a);
```

Evaluate the triple integral over the region `-Inf` $\leq x \leq 0$, $-100 \leq y \leq 0$, and $-100 \leq z \leq 0$.

```
format long
q1 = integral3(f,-Inf,0,-100,0,-100,0)
```

```
q1 =

     2.734244598320928e+03
```

Evaluate the integral again and specify accuracy to approximately 9 significant digits.

```
q2 = integral3(f,-Inf,0,-100,0,-100,0,'AbsTol', 0,'RelTol',1e-9)

q2 =

     2.734244599944285e+03
```

See Also

integral | integral2 | function_handle | trapz

Concepts

- “Parameterizing Functions”

Purpose

List custom interfaces exposed by COM server object

Syntax

```
customlist = h.interfaces
customlist = interfaces(h)
```

Description

`customlist = h.interfaces` returns cell array of strings `customlist` listing all custom interfaces implemented by the component in a specific COM server object. The server is designated by input argument `h`, the handle returned by the `actxcontrol` or `actxserver` function when creating that server.

`customlist = interfaces(h)` is an alternate syntax.

The `interfaces` function only lists the custom interfaces exposed by the object; it does not return interfaces. Use the `invoke` function to return a handle to a specific custom interface.

COM functions are available on Microsoft Windows systems only.

See Also

`actxcontrol` | `actxserver` | `invoke` | `get (COM)`

How To

- “Custom Interfaces”

Purpose

1-D data interpolation (table lookup)

Note If you pass non-uniformly spaced points and specify the `'v5cubic'` method, `interp1` will now issue a warning. In addition, the following syntaxes will be removed or changed in a future release:

- `interp1(...,'cubic')`
- `pp = interp1(...'pp')`
- `interp1(X,Y,Xq,[],...)`
- `interp1(...,*METHOD)`

For more information, and recommendations for updating your code, see "Interpolation and Computational Geometry Functionality Being Removed or Changed".

Syntax

```
yi = interp1(x,Y,xi)
yi = interp1(Y,xi)
yi = interp1(x,Y,xi,method)
yi = interp1(x,Y,xi,method,'extrap')
yi = interp1(x,Y,xi,method,extrapval)
pp = interp1(x,Y,method,'pp')
```

Description

`yi = interp1(x,Y,xi)` interpolates to find `yi`, the values of the underlying function `Y` at the points in the vector or array `xi`. `x` must be a vector. Y can be a scalar, a vector, or an array of any dimension, subject to the following conditions:

- If `Y` is a vector, it must have the same length as `x`. A scalar value for `Y` is expanded to have the same length as `x`. `xi` can be a scalar, a vector, or a multidimensional array, and `yi` has the same size as `xi`.
- If `Y` is an array that is not a vector, the size of `Y` must have the form `[n,d1,d2,...,dk]`, where `n` is the length of `x`. The interpolation is performed for each `d1`-by-`d2`-by-...-`dk` value in `Y`. The sizes of `xi` and `yi` are related as follows:

- If `xi` is a scalar or vector, `size(yi)` equals `[length(xi), d1, d2, ..., dk]`.
- If `xi` is an array of size `[m1,m2,...,mj]`, `yi` has size `[m1,m2,...,mj,d1,d2,...,dk]`.

`yi = interp1(Y,xi)` assumes that `x = 1:N`, where `N` is the length of `Y` for vector `Y`, or `size(Y,1)` for matrix `Y`.

`yi = interp1(x,Y,xi,method)` interpolates using alternative methods:

'nearest'	Nearest neighbor interpolation
'linear'	Linear interpolation (default)
'spline'	Cubic spline interpolation
'pchip'	Piecewise cubic Hermite interpolation
'cubic'	(Same as 'pchip')
'v5cubic'	Cubic interpolation used in MATLAB 5. This method does not extrapolate. Also, if `x` is not equally spaced, `'spline'` is used.

For the `'nearest'`, `'linear'`, and `'v5cubic'` methods, `interp1(x,Y,xi,method)` returns `NaN` for any element of `xi` that is outside the interval spanned by `x`. For all other methods, `interp1` performs extrapolation for out of range values.

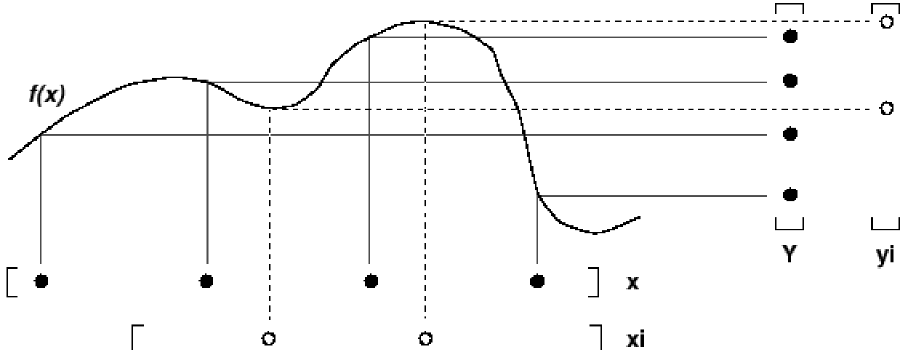
`yi = interp1(x,Y,xi,method,'extrap')` uses the specified interpolation algorithm specified by `method` to perform extrapolation for out of range values.

`yi = interp1(x,Y,xi,method,extrapval)` returns the scalar `extrapval` for out of range values. `NaN` and `0` are often used for `extrapval`.

`pp = interp1(x,Y,method,'pp')` uses the interpolation algorithm specified by `method` to generate the piecewise polynomial form (ppform) of `Y`. You can use any of the methods in the preceding table, except for

'v5cubic'. pp can then be evaluated via ppval. ppval(pp,xi) is the same as interp1(x,Y,xi,method,'extrap').

The interp1 command interpolates between data points. It finds values at intermediate points, of a one-dimensional function *f*(*x*) that underlies the data. This function is shown below, along with the relationship between vectors x, Y, xi, and yi.



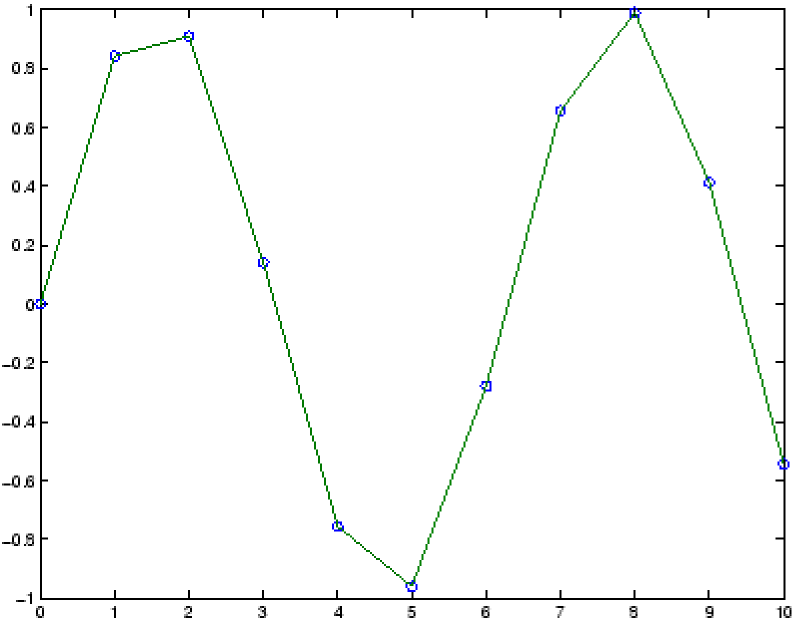
Interpolation is the same operation as *table lookup*. Described in table lookup terms, the *table* is [x,Y] and interp1 *looks up* the elements of xi in x, and, based upon their locations, returns values yi interpolated within the elements of Y.

Examples

Example 1

Generate a coarse sine curve and interpolate over a finer abscissa.

```
x = 0:10;
y = sin(x);
xi = 0:.25:10;
yi = interp1(x,y,xi);
plot(x,y,'o',xi,yi)
```



Example 2

Here are two vectors representing the census years from 1900 to 1990 and the corresponding United States population in millions of people.

```
t = 1900:10:1990;
p = [75.995  91.972  105.711  123.203  131.669...
     150.697  179.323  203.212  226.505  249.633];
```

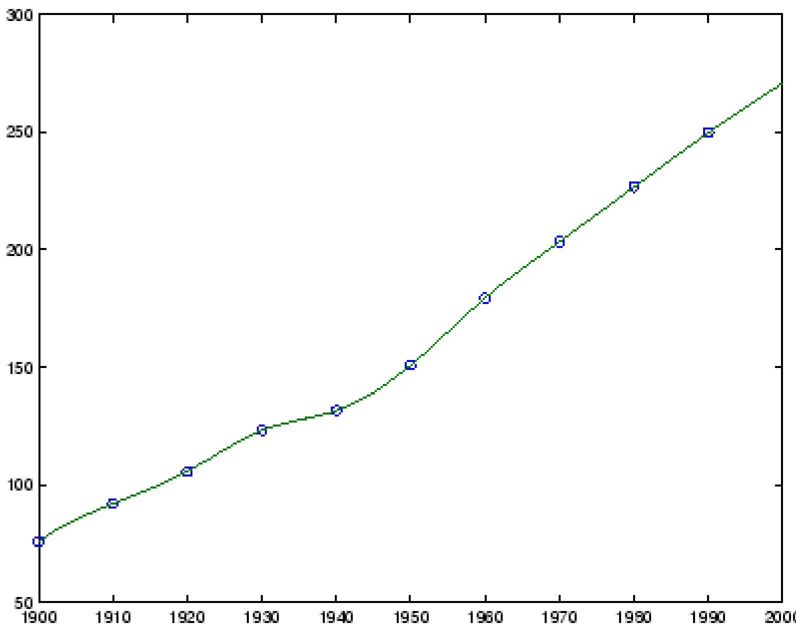
The expression `interp1(t,p,1975)` interpolates within the census data to estimate the population in 1975. The result is

```
ans =
    214.8585
```

Now interpolate within the data at every year from 1900 to 2000, and plot the result.

```
 x = 1900:1:2000;
```

```
y = interp1(t,p,x,'spline');
plot(t,p,'o',x,y)
```



Sometimes it is more convenient to think of interpolation in table lookup terms, where the data are stored in a single table. If a portion of the census data is stored in a single 5-by-2 table,

```
tab =
    1950    150.697
    1960    179.323
    1970    203.212
    1980    226.505
    1990    249.633
```

then the population in 1975, obtained by table lookup within the matrix `tab`, is

```
p = interp1(tab(:,1),tab(:,2),1975)
```

```
p =
    214.8585
```

Algorithms

The `interp1` command is written in MATLAB. The ’`nearest`' and ’`linear`' methods have straightforward implementations.

For the ’`spline`' method, `interp1` calls a function `spline` that uses the functions `ppval`, `mkpp`, and `unmkpp`. These routines form a small suite of functions for working with piecewise polynomials. The `spline` function uses them to perform the cubic spline interpolation. For access to more advanced features, see the `spline` reference page, the command-line help for these functions, and the Curve Fitting Toolbox™ spline functions.

For the `'pchip'` and `'cubic'` methods, `interp1` calls a function `pchip` that performs piecewise cubic interpolation within the vectors `x` and `y`. This method preserves monotonicity and the shape of the data. See the `pchip` reference page for more information.

References

[1] de Boor, C., *A Practical Guide to Splines*, Springer-Verlag, 1978.

See Also

`interpft` | `interp2` | `interp3` | `interpn` | `pchip` | `spline`

interp1q

Purpose

Quick 1-D linear interpolation

Note `interp1q` is not recommended. Use `interp1` instead.

Syntax

```
yi = interp1q(x,Y,xi)
```

Description

`yi = interp1q(x,Y,xi)` returns the value of the 1-D function `Y` at the points of column vector `xi` using linear interpolation. The vector `x` specifies the coordinates of the underlying interval. The length of output `yi` is equal to the length of `xi`.

`interp1q` is quicker than `interp1` on non-uniformly spaced data because it does no input checking.

For `interp1q` to work properly,

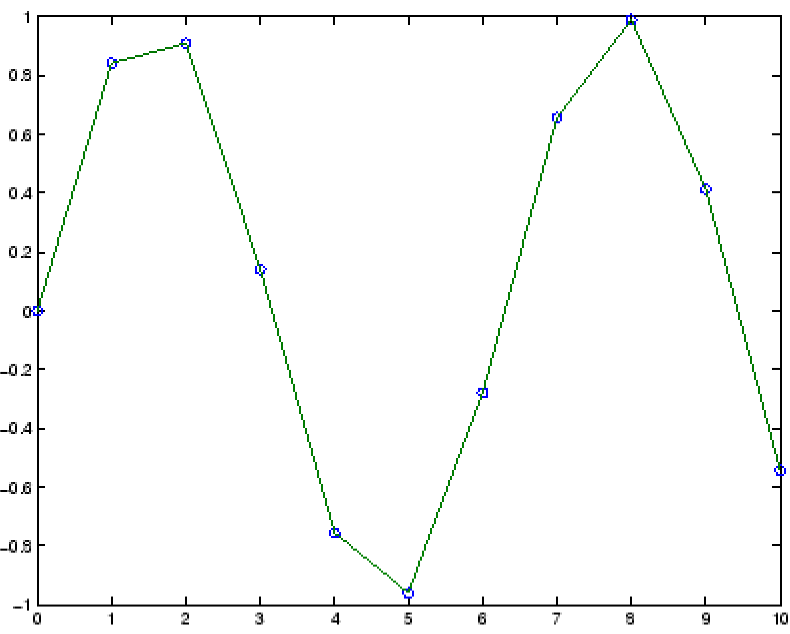
- `x` must be a monotonically increasing column vector.
- `Y` must be a column vector or matrix with `length(x)` rows.
- `xi` must be a column vector

`interp1q` returns NaN for any values of `xi` that lie outside the coordinates in `x`. If `Y` is a matrix, then the interpolation is performed for each column of `Y`, in which case `yi` is `length(xi)-by-size(Y,2)`.

Examples

Generate a coarse sine curve and interpolate over a finer abscissa.

```
x = (0:10)';
y = sin(x);
xi = (0:.25:10)';
yi = interp1q(x,y,xi);
plot(x,y,'o',xi,yi)
```



See Also

`interp1` | `interp2` | `interp3` | `interpn`

Purpose

Interpolation for 2-D gridded data in meshgrid format

Note In a future release, `interp2` will not accept mixed combinations of row and column vectors for the sample and query grids. For more information, and recommendations for updating your code, see "Interpolation and Computational Geometry Functionality Being Removed or Changed".

Syntax

```
ZI = interp2(X,Y,Z,XI,YI)
ZI = interp2(Z,XI,YI)
ZI = interp2(Z,ntimes)
ZI = interp2(X,Y,Z,XI,YI,method)
ZI = interp2(...,method, extrapval)
```

Description

`ZI = interp2(X,Y,Z,XI,YI)` returns matrix `ZI` containing elements corresponding to the elements of `XI` and `YI` and determined by interpolation within the two-dimensional function specified by matrices `X`, `Y`, and `Z`. `X` and `Y` must be monotonic, and have the same format ("plaid") as if they were produced by `meshgrid`. Matrices `X` and `Y` specify the points at which the data `Z` is given. Out of range values are returned as `NaN`s.

`XI` and `YI` can be matrices, in which case `interp2` returns the values of `Z` corresponding to the points `(XI(i,j),YI(i,j))`. Alternatively, you can pass in the row and column vectors `xi` and `yi`, respectively. In this case, `interp2` interprets these vectors as if you issued the command `meshgrid(xi,yi)`.

`ZI = interp2(Z,XI,YI)` assumes that `X = 1:n` and `Y = 1:m`, where `[m,n] = size(Z)`.

`ZI = interp2(Z,ntimes)` expands `Z` by interleaving interpolates between every element, working recursively for `ntimes`. `interp2(Z)` is the same as `interp2(Z,1)`.

`ZI = interp2(X,Y,Z,XI,YI,method)` specifies an alternative interpolation method:

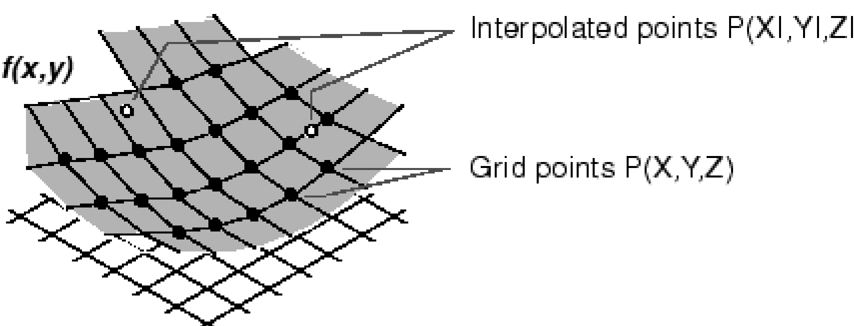
`'nearest'`	Nearest neighbor interpolation
`'linear'`	Linear interpolation (default)
`'spline'`	Cubic spline interpolation. This method returns extrapolated values when you query points outside the domain of `X` and `Y`.
`'cubic'`	Cubic interpolation, as long as data is uniformly-spaced. `interp2` switches to `'spline'` and issues a warning if the data is not equally spaced.

All interpolation methods require that `X` and `Y` be monotonic, and have the same format ("plaid") as if they were produced by `meshgrid`. If you provide two monotonic vectors, `interp2` changes them to a plaid internally. Variable spacing is handled by mapping the given values in `X`, `Y`, `XI`, and `YI` to an equally spaced domain before interpolating.

`ZI = interp2(...,method, extrapval)` specifies a method and a scalar value for `ZI` outside of the domain created by `X` and `Y`. Thus, `ZI` equals `extrapval` for any value of `YI` or `XI` that is not spanned by `Y` or `X` respectively. A method must be specified to use `extrapval`. The default method is `'linear'`.

Tips

The `interp2` command interpolates between data points. It finds values of a two-dimensional function $f(x,y)$ underlying the data at intermediate points.



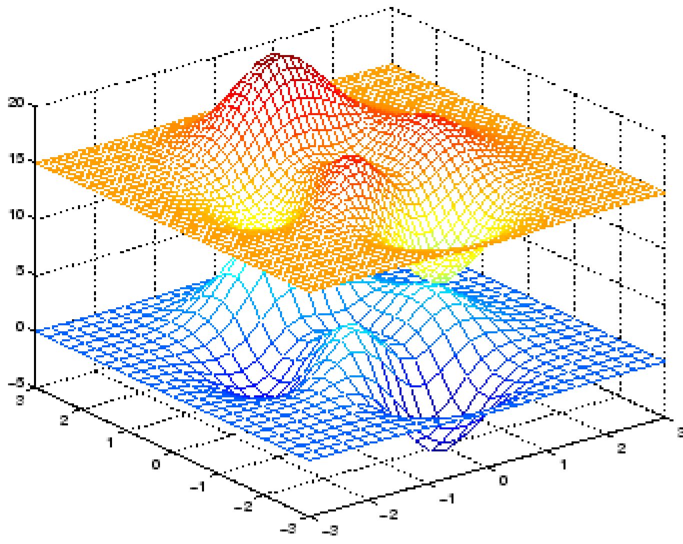
Interpolation is the same operation as table lookup. Described in table lookup terms, the table is `tab = [NaN,Y;X,Z]` and `interp2` looks up the elements of `XI` in `X`, `YI` in `Y`, and, based upon their location, returns values `ZI` interpolated within the elements of `Z`.

Examples

Example 1

Interpolate the `peaks` function over a finer grid.

```
[X,Y] = meshgrid(-3:.25:3);
Z = peaks(X,Y);
[XI,YI] = meshgrid(-3:.125:3);
ZI = interp2(X,Y,Z,XI,YI);
mesh(X,Y,Z), hold, mesh(XI,YI,ZI+15)
hold off
axis([-3 3 -3 3 -5 20])
```



Example 2

Given this set of employee data,

```
years = 1950:10:1990;
service = 10:10:30;
wage = [150.697 199.592 187.625
179.323 195.072 250.287
203.212 179.092 322.767
226.505 153.706 426.730
249.633 120.281 598.243];
```

it is possible to interpolate to find the wage earned in 1975 by an employee with 15 years' service:

```
[X,Y] = meshgrid(service,years');
w = interp2(X,Y,wage,15,1975)
w =
  190.6288
```

See Also

griddata | interp1 | interp3 | interpn | meshgrid

interp3

Purpose

Interpolation for 3-D gridded data in meshgrid format

Note In a future release, `interp3` will not accept mixed combinations of row and column vectors for the sample and query grids. For more information, and recommendations for updating your code, see "Interpolation and Computational Geometry Functionality Being Removed or Changed".

Syntax

```
VI = interp3(X,Y,Z,V,XI,YI,ZI)
VI = interp3(V,XI,YI,ZI)
VI = interp3(V,ntimes)
VI = interp3(...,method)
VI = interp3(...,method,extrapval)
```

Description

`VI = interp3(X,Y,Z,V,XI,YI,ZI)` interpolates to find `VI`, the values of the underlying three-dimensional function `V` at the points in arrays `XI`, `YI` and `ZI`. `XI`,`YI`, `ZI` must be arrays of the same size, or vectors. Vector arguments that are not the same size, and have mixed orientations (i.e. with both row and column vectors) are passed through `meshgrid` to create the `Y1`, `Y2`, `Y3` arrays. Arrays `X`, `Y`, and `Z` specify the points at which the data `V` is given. Out of range values are returned as `NaN`.

`VI = interp3(V,XI,YI,ZI)` assumes `X=1:N`, `Y=1:M`, `Z=1:P` where `[M,N,P]=size(V)`.

`VI = interp3(V,ntimes)` expands `V` by interleaving interpolates between every element, working recursively for `ntimes` iterations. The command `interp3(V)` is the same as `interp3(V,1)`.

`VI = interp3(...,method)` specifies alternative methods:

'nearest'	Nearest neighbor interpolation
'linear'	Linear interpolation (default)

'spline'	Cubic spline interpolation. This method returns extrapolated values when you query points outside the domain of X, Y, and Z.
'cubic'	Cubic interpolation, as long as data is uniformly-spaced. interp3 switches to 'spline' and issues a warning if the data is not equally spaced.

VI = interp3(...,method,extrapval) specifies a method and a value for VI outside of the domain created by X, Y and Z. Thus, VI equals extrapval for any value of XI, YI or ZI that is not spanned by X, Y, and Z, respectively. You must specify a method to use extrapval. The default method is 'linear'.

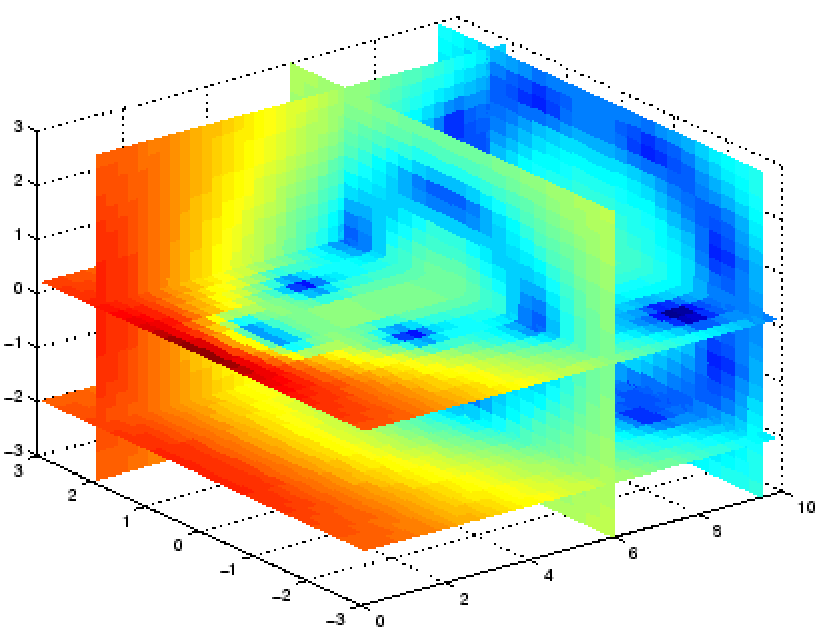
Discussion

All the interpolation methods require that X,Y and Z be monotonic and have the same format ("plaid") as if they were created using meshgrid. X, Y, and Z can be non-uniformly spaced.

Examples

To generate a coarse approximation of flow and interpolate over a finer mesh:

```
[x,y,z,v] = flow(10);
[xi,yi,zi] = meshgrid(.1:.25:10, -3:.25:3, -3:.25:3);
vi = interp3(x,y,z,v,xi,yi,zi); % vi is 25-by-40-by-25
slice(xi,yi,zi,vi,[6 9.5],2,[-2 .2]), shading flat
```



See Also

`interp1` | `interp2` | `interpn` | `meshgrid`

Purpose

1-D interpolation using FFT method

Syntax

```
y = interpft(x,n)
y = interpft(x,n,dim)
```

Description

`y = interpft(x,n)` returns the vector `y` that contains the value of the periodic function `x` resampled to `n` equally spaced points.

If `length(x) = m`, and `x` has sample interval `dx`, then the new sample interval for `y` is `dy = dx*m/n`. Note that `n` cannot be smaller than `m`.

If `X` is a matrix, `interpft` operates on the columns of `X`, returning a matrix `Y` with the same number of columns as `X`, but with `n` rows.

`y = interpft(x,n,dim)` operates along the specified dimension.

Algorithms

The `interpft` command uses the FFT method. The original vector `x` is transformed to the Fourier domain using `fft` and then transformed back with more points.

Examples

Interpolate a triangle-like signal using an interpolation factor of 5. First, set up signal to be interpolated:

```
y = [0 .5 1 1.5 2 1.5 1 .5 0 -.5 -1 -1.5 -2 -1.5 -1 -.5 0];
N = length(y);
```

Perform the interpolation:

```
L = 5;
M = N*L;
x = 0:L:L*N-1;
xi = 0:M-1;
yi = interpft(y,M);
plot(x,y,'o',xi,yi,'*')
legend('Original data','Interpolated data')
```

See Also

`interp1`

interpn

Purpose

Interpolation for 1-D, 2-D, 3-D, and N-D gridded data in ndgrid format

---

Note In a future release, `interpn` will not accept mixed combinations of row and column vectors for the sample and query grids. For more information, and recommendations for updating your code, see "Interpolation and Computational Geometry Functionality Being Removed or Changed".

Syntax

```
VI = interpn(X1,X2,X3,...,V,Y1,Y2,Y3,...)
VI = interpn(V,Y1,Y2,Y3,...)
VI = interpn(V,ntimes)
VI = interpn(...,method)
VI = interpn(...,method,extrapval)
```

Description

`VI = interpn(X1,X2,X3,...,V,Y1,Y2,Y3,...)` interpolates to find `VI`, the values of the underlying multidimensional function `V` at the points in the arrays `Y1`, `Y2`, `Y3`, etc. For an n-dimensional array `V`, `interpn` is called with `2*N+1` arguments. Arrays `X1`, `X2`, `X3`, etc. specify the points at which the data `V` is given. Out of range values are returned as `NaN`s. `Y1`, `Y2`, `Y3`, etc. must be arrays of the same size, or vectors. Vector arguments that are not the same size, and have mixed orientations (i.e. with both row and column vectors) are passed through `ndgrid` to create the `Y1`, `Y2`, `Y3`, etc. arrays. `interpn` works for all n-dimensional arrays with 2 or more dimensions.

`VI = interpn(V,Y1,Y2,Y3,...)` interpolates as above, assuming `X1 = 1:size(V,1)`, `X2 = 1:size(V,2)`, `X3 = 1:size(V,3)`, etc.

`VI = interpn(V,ntimes)` expands `V` by interleaving interpolates between each element, working recursively for `ntimes` iterations. `interpn(V)` is the same as `interpn(V,1)`.

`VI = interpn(...,method)` specifies alternative methods:

'nearest'	Nearest neighbor interpolation
'linear'	Linear interpolation (default)
'spline'	Cubic spline interpolation. This method returns extrapolated values when you query points outside the domain of X1,...,Xn.
'cubic'	Cubic interpolation, as long as data is uniformly-spaced. interpn switches to 'spline' and issues a warning if the data is not equally spaced.

VI = interpn(...,method,extrapval) specifies a method and a value for VI outside of the domain created by X1, X2, ... Thus, VI equals extrapval for any value of Y1, Y2,... that is not spanned by X1, X2,... respectively. You must specify a method to use extrapval. The default method is 'linear'.

interpn requires that X1, X2, X3, ... be monotonic and plaid (as if they were created using ndgrid). X1, X2, X3, and so on can be non-uniformly spaced.

Discussion

All the interpolation methods require that X1,X2, X3 ... be monotonic and have the same format ("plaid") as if they were created using ndgrid. X1,X2,X3,... and Y1, Y2, Y3, etc. can be non-uniformly spaced.

Examples

Start by defining an anonymous function to compute $f = te^{-x^2-y^2-z^2}$:

```
f = @(x,y,z,t) t.*exp(-x.^2 - y.^2 - z.^2);
```

Build the lookup table by evaluating the function f on a grid constructed by ndgrid:

```
[x,y,z,t] = ndgrid(-1:0.2:1,-1:0.2:1,-1:0.2:1,0:2:10);
v = f(x,y,z,t);
```

Now construct a finer grid:

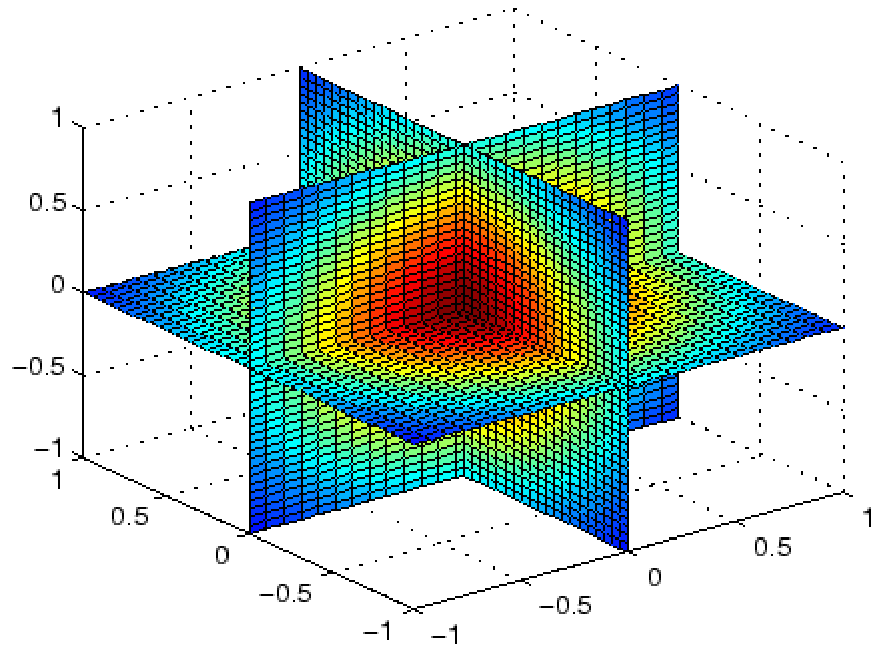
```
[xi,yi,zi,ti] = ndgrid(-1:0.05:1,-1:0.08:1,-1:0.05:1, ...
                       0:0.5:10);
```

Compute the spline interpolation at `xi`, `yi`, `zi`, and `ti`:

```
vi = interpn(x,y,z,t,v,xi,yi,zi,ti,'spline');
```

Plot the interpolated function, and then create a movie from the plot:

```
nframes = size(ti, 4);
for j = 1:nframes
   slice(yi(:,:,:,j), xi(:,:,:,j), zi(:,:,:,j), ...
         vi(:,:,:,j),0,0,0);
   caxis([0 10]);
   M(j) = getframe;
end
movie(M);
```



See Also

`interp1` | `interp2` | `interp3` | `ndgrid`

interpstreamspeed

Purpose

Interpolate stream-line vertices from flow speed

Syntax

```
interpstreamspeed(X,Y,Z,U,V,W,vertices)
interpstreamspeed(U,V,W,vertices)
interpstreamspeed(X,Y,Z,speed,vertices)
interpstreamspeed(speed,vertices)
interpstreamspeed(X,Y,U,V,vertices)
interpstreamspeed(U,V,vertices)
interpstreamspeed(X,Y,speed,vertices)
interpstreamspeed(speed,vertices)
interpstreamspeed(...,sf)
vertsout = interpstreamspeed(...)
```

Description

`interpstreamspeed(X,Y,Z,U,V,W,vertices)` interpolates streamline vertices based on the magnitude of the vector data `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`.

`interpstreamspeed(U,V,W,vertices)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(U)`.

`interpstreamspeed(X,Y,Z,speed,vertices)` uses the 3-D array `speed` for the speed of the vector field.

`interpstreamspeed(speed,vertices)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p]=size(speed)`.

`interpstreamspeed(X,Y,U,V,vertices)` interpolates streamline vertices based on the magnitude of the vector data `U`, `V`.

The arrays X and Y, which define the coordinates for U and V, must be monotonic, but do not need to be uniformly spaced. X and Y must have the same number of elements, as if produced by meshgrid.

interpstreamspeed(U,V,vertices) assumes X and Y are determined by the expression

```
[X Y] = meshgrid(1:n,1:m)
```

where [M N]=size(U).

interpstreamspeed(X,Y,speed,vertices) uses the 2-D array speed for the speed of the vector field.

interpstreamspeed(speed,vertices) assumes X and Y are determined by the expression

```
[X Y] = meshgrid(1:n,1:m)
```

where [M,N]= size(speed).

interpstreamspeed(...,sf) uses sf to scale the magnitude of the vector data and therefore controls the number of interpolated vertices. For example, if sf is 3, then interpstreamspeed creates only one-third of the vertices.

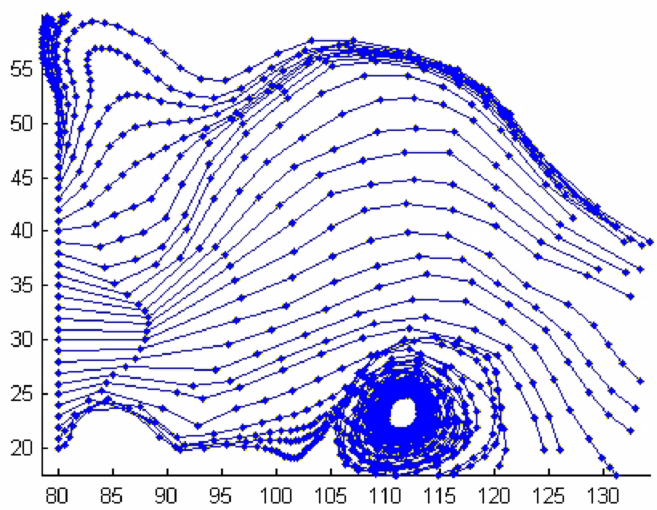
vertsout = interpstreamspeed(...) returns a cell array of vertex arrays.

Examples

This example draws streamlines using the vertices returned by interpstreamspeed. Dot markers indicate the location of each vertex. This example enables you to visualize the relative speeds of the flow data. Streamlines having widely spaced vertices indicate faster flow; those with closely spaced vertices indicate slower flow.

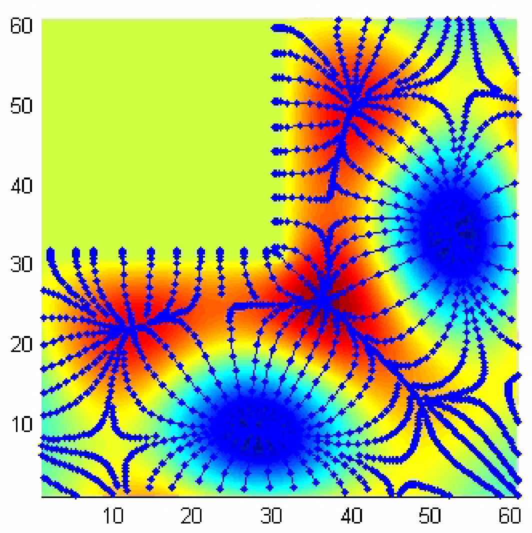
```
load wind
[sx sy sz] = meshgrid(80,20:1:55,5);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
iverts = interpstreamspeed(x,y,z,u,v,w,verts,.2);
sl = streamline(iverts);
```

```
set(sl,'Marker','.')
axis tight; view(2); daspect([1 1 1])
```



This example plots streamlines whose vertex spacing indicates the value of the gradient along the streamline.

```
z = membrane(6,30);
[u v] = gradient(z);
[verts averts] = streamslice(u,v);
iverts = interpstreamspeed(u,v,verts,15);
sl = streamline(iverts);
set(sl,'Marker','.')
hold on; pcolor(z); shading interp
axis tight; view(2); daspect([1 1 1])
```



See Also

stream2 | stream3 | streamline | streamslice | streamparticles

intersect

Purpose

Set intersection of two arrays

Syntax

```
C = intersect(A,B)
C = intersect(A,B,'rows')
[C,ia,ib] = intersect(A,B)
[C,ia,ib] = intersect(A,B,'rows')

[C,ia,ib] = intersect(___,setOrder)

[C,ia,ib] = intersect(A,B,'legacy')

[C,ia,ib] = intersect(A,B,'rows','legacy')
```

Description

`C = intersect(A,B)` returns the values common to both `A` and `B`. The values of `C` are in sorted order.

`C = intersect(A,B,'rows')` treats each row of `A` and each row of `B` as single entities and returns the rows common to both `A` and `B`. The rows of the matrix `C` are in sorted order.

The `'rows'` option does not support cell arrays.

`[C,ia,ib] = intersect(A,B)` also returns index vectors `ia` and `ib`, such that `C = A(ia)` and `C = B(ib)`.

`[C,ia,ib] = intersect(A,B,'rows')` also returns index vectors `ia` and `ib`, such that `C = A(ia,:)` and `C = B(ib,:)`.

`[C,ia,ib] = intersect(___,setOrder)` returns `C` in a specific order using any of the input arguments in the previous syntaxes. `setOrder='sorted'` returns the values (or rows) of `C` in sorted order. `setOrder='stable'` returns the values (or rows) of `C` in the same order as `A`, then `B`. If no value is specified, the default is `'sorted'`.

`[C,ia,ib] = intersect(A,B,'legacy')` and `[C,ia,ib] = intersect(A,B,'rows','legacy')` preserve the behavior of the `intersect` function from R2012b and prior releases.

Input Arguments

A,B - Input arrays

vectors | matrices | N-D arrays

Input arrays, specified as vectors, matrices, or N-D arrays whose elements can be any numeric, `logical`, or `char` data type. `A` and `B` also can be cell arrays of strings. Furthermore, `A` and `B` can be objects with the following class methods:

- `sort` (or `sortrows` for the `'rows'` option), where the second output of this method is `double` or another built-in numeric class
- `ne`

These objects include heterogeneous arrays derived from the same root class.

`A` and `B` must be of the same class with the following exceptions:

- `logical`, `char`, and all numeric classes can combine with `double` arrays.
- Cell arrays of strings can combine with `char` arrays.

If you specify the `'rows'` option, `A` and `B` must have the same number of columns.

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

setOrder - Order flag

`'sorted'` (default) | `'stable'`

Order flag, specified as `'sorted'` or `'stable'`, indicates the order of the values (or rows) in `C`.

Order Flag	Meaning
'sorted'	The values (or rows) in C return in sorted order. For example: C = intersect([7 0 5],[7 1 5],'sorted') returns C = [5 7].
'stable'	The values (or rows) in C return in the same order as they appear in A and B. For example: C = intersect([7 0 5],[7 1 5],'stable') returns C = [7 5].

Data Types
char

Output Arguments

C - Values common to A and B

vector | matrix

Values common to A and B, returned as a vector or matrix. The following describes the shape of C when the 'legacy' flag is not specified:

- If the 'rows' flag is not specified, then C is a column vector unless both A and B are row vectors.
- If the 'rows' flag is not specified and both A and B are row vectors, then C is a row vector.
- If the 'rows' flag is specified, then C is a matrix containing the rows in common from A and B.
- If A and B have no values (or rows) in common, then C is an empty matrix.

The class of the inputs A and B determines the class of C:

- If the class of A and B are the same, then C is the same class.

- If you combine a `char` or nondouble numeric class with `double`, then `C` is the same class as the nondouble input.
- If you combine a `logical` class with `double`, then `C` is `double`.
- If you combine a cell array of strings with `char`, then `C` is a cell array of strings.

ia - Index to A

column vector

Index to `A`, returned as a column vector when the `'legacy'` flag is not specified. `ia` identifies the values (or rows) in `A` that are common to `B`. If there is a repeated value (or row) in `A`, then `ia` contains the index to the first occurrence of the value (or row).

ib - Index to B

column vector

Index to `B`, returned as a column vector when the `'legacy'` flag is not specified. `ib` identifies the values (or rows) in `B` that are common to `A`. If there is a repeated value (or row) in `B`, then `ib` contains the index to the first occurrence of the value (or row).

Examples

Intersection of Two Vectors

Define two vectors with values in common.

```
A = [7 1 7 7 4]; B = [7 0 4 4 0];
```

Find the values common to both `A` and `B`.

```
C = intersect(A,B)
```

```
C =

     4     7
```

Intersection of Two Vectors and Their Indices

Define two vectors with values in common.

```
A = [7 1 7 7 4]; B = [7 0 4 4 0];
```

Find the values common to both `A` and `B` as well as the index vectors `ia` and `ib`, such that `C = A(ia)` and `C = B(ib)`.

```
[C,ia,ib] = intersect(A,B)
```

```
C =

     4     7


ia =

     5
     1


ib =

     3
     1
```

Intersection of Rows in Two Matrices

Define two matrices with rows in common.

```
A = [2 2 2; 0 0 1; 1 2 3; 1 1 1];
B = [1 2 3; 2 2 2; 2 2 0];
```

Find the rows common to both `A` and `B` as well as the index vectors `ia` and `ib`, such that `C = A(ia,:)` and `C = B(ib,:)`.

```
[C,ia,ib] = intersect(A,B,'rows')
```

```
C =

     1     2     3
     2     2     2


ia =

     3
     1


ib =

     1
     2
```

A and B do not need to have the same number of rows, but they must have the same number of columns.

Intersection with Specified Output Order

Use the `setOrder` argument to specify the ordering of the values in C.

Specify `'stable'` if you want the values in C to have the same order as in A.

```
A = [7 1 7 7 4]; B = [7 0 4 4 0];
[C,ia,ib] = intersect(A,B,'stable')

C =

     7     4


ia =

     1
     5
```

```
ib =

     1
     3
```

Alternatively, you can specify `'sorted'` order.

```
[C,ia,ib] = intersect(A,B,'sorted')
```

```
C =

     4     7


ia =

     5
     1


ib =

     3
     1
```

Intersection of Vectors Containing NaNs

Define two vectors containing `NaN`.

```
A = [5 NaN NaN]; B = [5 NaN NaN];
```

Find the values common to both `A` and `B`.

```
C = intersect(A,B)
```

```
C =
```

```
     5
```

`intersect` treats `NaN` values as distinct.

Cell Array of Strings with Trailing White Space

Create a cell array of strings, `A`.

```
A = {'dog','cat','fish','horse'};
```

Create a cell array of strings, `B`, where some of the strings have trailing white space.

```
B = {'dog ','cat','fish ','horse'};
```

Find the strings common to both `A` and `B`.

```
[C,ia,ib] = intersect(A,B)
```

```
C =

    'cat'    'horse'


ia =

     2
     4


ib =

     2
     4
```

`intersect` treats trailing white space in cell arrays of strings as distinct characters.

Intersection of Arrays of Different Classes and Shapes

Create a column vector character array.

```
A = ['A';'B';'C'], class(A)
```

```
A =

A
B
C


ans =

char
```

Create a 2-by-3 matrix containing elements of numeric type `double`.

```
B = [65 66 67;68 69 70], class(B)
```

```
B =

    65    66    67
    68    69    70


ans =

double
```

Find the values common to both `A` and `B`.

```
[C,ia,ib] = intersect(A,B)
```

```
C =

A
B
C


ia =

     1
     2
     3


ib =

     1
     3
     5
```

`intersect` interprets `B` as a character array and returns a character array, `C`.

```
class(C)

ans =

char
```

Intersection of Char and Cell Array of Strings

Create a character array containing animal names that have three letters.

```
A = ['dog';'cat';'fox';'pig'];
class(A)

ans =
```

```
char
```

Create a cell array of strings containing animal names of varying lengths.

```
B = {'cat','dog','fish','horse'};
class(B)

ans =

cell
```

Find the strings common to both `A` and `B`.

```
C = intersect(A,B)

C =

    'cat'
    'dog'
```

The result, `C`, is a cell array of strings.

```
class(C)

ans =

cell
```

Preserve Legacy Behavior of `intersect`

Use the `'legacy'` flag to preserve the behavior of `intersect` from R2012b and prior releases in your code.

Find the intersection of `A` and `B` with the current behavior.

```
A = [7 1 7 7 4]; B = [7 0 4 4 0];
```

```
[C1,ia1,ib1] = intersect(A,B)

C1 =

     4     7


ia1 =

     5
     1


ib1 =

     3
     1
```

Find the unique elements of A and preserve the legacy behavior.

```
[C2,ia2,ib2] = intersect(A,B,'legacy')

C2 =

     4     7


ia2 =

     5     4


ib2 =

     4     1
```

See Also

unique | union | ismember | issorted | setdiff | setxor | sort

intmax

Purpose

Largest value of specified integer type

Syntax

```
v = intmax
v = intmax('classname')
```

Description

`v = intmax` is the largest positive value that can be represented in the MATLAB software with a 32-bit integer. Any value larger than the value returned by `intmax` saturates to the `intmax` value when cast to a 32-bit integer.

`v = intmax('classname')` is the largest positive value in the integer class `classname`. Valid values for the string `classname` are

'int8'	'int16'	'int32'	'int64'
'uint8'	'uint16'	'uint32'	'uint64'

`intmax('int32')` is the same as `intmax` with no arguments.

Examples

Find the maximum value for a 64-bit signed integer:

```
v = intmax('int64')
v =
  9223372036854775807
```

Convert this value to a 32-bit signed integer:

```
x = int32(v)
x =
  2147483647
```

Compare the result with the default value returned by `intmax`:

```
isequal(x, intmax)
ans =
     1
```

See Also

`intmin` | `realmax` | `realmin` | `int8` | `uint8` | `isa` | `class`

Purpose

Smallest value of specified integer type

Syntax

```
v = intmin
v = intmin('classname')
```

Description

`v = intmin` is the smallest value that can be represented in the MATLAB software with a 32-bit integer. Any value smaller than the value returned by `intmin` saturates to the `intmin` value when cast to a 32-bit integer.

`v = intmin('classname')` is the smallest positive value in the integer class `classname`. Valid values for the string `classname` are

'int8'	'int16'	'int32'	'int64'
'uint8'	'uint16'	'uint32'	'uint64'

`intmin('int32')` is the same as `intmin` with no arguments.

Examples

Find the minimum value for a 64-bit signed integer:

```
v = intmin('int64')
v =
 -9223372036854775808
```

Convert this value to a 32-bit signed integer:

```
x = int32(v)
x =
  2147483647
```

Compare the result with the default value returned by `intmin`:

```
isequal(x, intmin)
ans =
     1
```

See Also

`intmax` | `realmin` | `realmax` | `int8` | `uint8` | `isa` | `class`

Purpose

Matrix inverse

Syntax

```
Y = inv(X)
```

Description

`Y = inv(X)` returns the inverse of the square matrix `X`. A warning message is printed if `X` is badly scaled or nearly singular.

In practice, it is seldom necessary to form the explicit inverse of a matrix. A frequent misuse of `inv` arises when solving the system of linear equations $Ax = b$. One way to solve this is with `x = inv(A)*b`. A better way, from both an execution time and numerical accuracy standpoint, is to use the matrix division operator `x = A\b`. This produces the solution using Gaussian elimination, without forming the inverse. See `mldivide` (\) for further information.

Examples

Here is an example demonstrating the difference between solving a linear system by inverting the matrix with `inv(A)*b` and solving it directly with `A\b`. A random matrix `A` of order 500 is constructed so that its condition number, `cond(A)`, is `1.e10`, and its norm, `norm(A)`, is `1`. The exact solution `x` is a random vector of length 500 and the right-hand side is `b = A*x`. Thus the system of linear equations is badly conditioned, but consistent.

On a 300 MHz, laptop computer the statements

```
n = 500;
Q = orth(randn(n,n));
d = logspace(0,-10,n);
A = Q*diag(d)*Q';
x = randn(n,1);
b = A*x;
tic, y = inv(A)*b; toc
err = norm(y-x)
res = norm(A*y-b)
```

produce

```
elapsed_time =
```

```
    1.4320
err =
    7.3260e-006
res =
    4.7511e-007
```

while the statements

```
tic, z = A\b, toc
err = norm(z-x)
res = norm(A*z-b)
```

produce

```
elapsed_time =
    0.6410
err =
    7.1209e-006
res =
    4.4509e-015
```

It takes almost two and one half times as long to compute the solution with `y = inv(A)*b` as with `z = A\b`. Both produce computed solutions with about the same error, `1.e-6`, reflecting the condition number of the matrix. But the size of the residuals, obtained by plugging the computed solution back into the original equations, differs by several orders of magnitude. The direct solution produces residuals on the order of the machine accuracy, even though the system is badly conditioned.

The behavior of this example is typical. Using `A\b` instead of `inv(A)*b` is two to three times as fast and produces residuals on the order of machine accuracy, relative to the magnitude of the data.

See Also

`det` | `lu` | `rref` | `mldivide` | `Arithmetic Operators \,/`

Purpose

Inverse of Hilbert matrix

Syntax

```
H = invhilb(n)
```

Description

`H = invhilb(n)` generates the exact inverse of the exact Hilbert matrix for `n` less than about 15. For larger `n`, `invhilb(n)` generates an approximation to the inverse Hilbert matrix.

Limitations

The exact inverse of the exact Hilbert matrix is a matrix whose elements are large integers. These integers may be represented as floating-point numbers without roundoff error as long as the order of the matrix, `n`, is less than 15.

Comparing `invhilb(n)` with `inv(hilb(n))` involves the effects of two or three sets of roundoff errors:

- The errors caused by representing `hilb(n)`
- The errors in the matrix inversion process
- The errors, if any, in representing `invhilb(n)`

It turns out that the first of these, which involves representing fractions like 1/3 and 1/5 in floating-point, is the most significant.

Examples

`invhilb(4)` is

```
    16     -120      240     -140
  -120     1200    -2700     1680
   240    -2700     6480    -4200
  -140     1680    -4200     2800
```

References

[1] Forsythe, G. E. and C. B. Moler, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, 1967, Chapter 19.

See Also

`hilb`

Purpose

Invoke method on COM object or interface, or display methods

Syntax

```
S = h.invoke
S = h.invoke('methodname')
S = h.invoke('methodname',arg1,arg2, ...)
S = h.invoke('custominterfacename')
S = invoke(h,...)
```

Description

`S = h.invoke` returns structure array `S` containing a list of all methods supported by the object or interface, `h`, along with the prototypes for these methods. If `S` is empty, either there are no properties or methods in the object, or the MATLAB software cannot read the object's type library. Refer to the COM vendor's documentation.

`S = h.invoke('methodname')` invokes the method specified in the string `methodname`, and returns an output value, if any, in `S`. The data type of the return value depends on the invoked method, which is determined by the control or server.

`S = h.invoke('methodname',arg1,arg2, ...)` invokes the method specified in the string `methodname` with input arguments `arg1`, `arg2`, etc.

`S = h.invoke('custominterfacename')` returns an `Interface` object `S`, which is a handle to a custom interface implemented by the COM component. The `h` argument is a handle to the COM object. The `custominterfacename` argument is a string returned by the `interfaces` function.

`S = invoke(h,...)` is an alternate syntax. For Automation objects, if the vendor provides documentation for specific properties or methods, use the `S = invoke(h, ...)` syntax to call them.

If the method returns a COM interface, then `invoke` returns a new MATLAB COM object that represents the interface returned. For a description of how MATLAB converts COM types, see "Handling COM Data in MATLAB Software".

COM functions are available on Microsoft Windows systems only.

Examples

Invoke the Redraw method in the mwsamp control:

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.1', [0 0 200 200], f);
h.Radius = 100;
h.invoke('Redraw');
```

Call the method directly:

```
h.Redraw;
```

Display all mwsamp methods:

```
h.invoke
```

MATLAB displays (in part):

```
ans =
   AboutBox = void AboutBox(handle)
   Beep = void Beep(handle)
   FireClickEvent = void FireClickEvent(handle)
             .
             .
```

Getting a Custom Interface Example

Once you have created a COM server, you can query the server component to see if any custom interfaces are implemented. Use the interfaces function to return a list of all available custom interfaces:

```
h = actxserver('mytestenv.calculator');
customlist = h.interfaces
```

MATLAB displays:

```
customlist =
   ICalc1
```

```
ICalc2
ICalc3
```

To get a handle to the custom interface you want, use the `invoke` function, specifying the handle returned by `actxcontrol` or `actxserver` and also the name of the custom interface:

```
c1 = h.invoke('ICalc1')
```

MATLAB displays:

```
c1 =
   Interface.Calc_1.0_Type_Library.ICalc_Interface
```

You can now use this handle with most of the COM client functions to access the properties and methods of the object through the selected custom interface.

See Also

`methods` | `ismethod` | `interfaces`

How To

- "Handling COM Data in MATLAB Software"
- "Custom Interfaces"

ipermute

Purpose

Inverse permute dimensions of N-D array

Syntax

```
A = ipermute(B,order)
```

Description

`A = ipermute(B,order)` is the inverse of `permute`. `ipermute` rearranges the dimensions of `B` so that `permute(A,order)` will produce `B`. `B` has the same values as `A` but the order of the subscripts needed to access any particular element are rearranged as specified by `order`. All the elements of `order` must be unique.

Tips

`permute` and `ipermute` are a generalization of transpose (`.'`) for multidimensional arrays.

Examples

Consider the 2-by-2-by-3 array `a`:

```
a = cat(3,eye(2),2*eye(2),3*eye(2))

a(:,:,1) =                a(:,:,2) =
     1     0                   2     0
     0     1                   0     2

a(:,:,3) =
     3     0
     0     3
```

Permuting and inverse permuting `a` in the same fashion restores the array to its original form:

```
B = permute(a,[3 2 1]);
C = ipermute(B,[3 2 1]);
isequal(a,C)
ans=

   1
```

See Also

`permute`

Purpose

Detect state

Description

These functions detect the state of MATLAB entities:

`isa`	Detect object of given MATLAB class or Java class
`isappdata`	Determine if object has specific application-defined data
`iscell`	Determine if input is cell array
`iscellstr`	Determine if input is cell array of strings
`ischar`	Determine if input is character array
`iscom`	Determine if input is Component Object Model (COM) object
`isdir`	Determine if input is folder
`isempty`	Determine if input is empty array
`isequal`	Determine if arrays are numerically equal
`isequaln`	Determine if arrays are numerically equal, treating `NaN`s as equal
`isevent`	Determine if input is Component Object Model (COM) object event
`isfield`	Determine if input is MATLAB structure array field
`isfinite`	Detect finite elements of array
`isfloat`	Determine if input is floating-point array
`isglobal`	Determine if input is global variable
`ishandle`	Detect valid graphics object handles
`ishold`	Determine if graphics hold state is on
`isinf`	Detect infinite elements of array
`isinteger`	Determine if input is integer array

isinterface	Determine if input is Component Object Model (COM) interface
isjava	Determine if input is Java object
iskeyword	Determine if input is MATLAB keyword
isletter	Detect elements that are alphabetic letters
islogical	Determine if input is logical array
ismac	Determine if running MATLAB for Macintosh OS X platform
ismember	Detect members of specific set
ismethod	Determine if input is object method
isnan	Detect elements of array that are not a number (NaN)
isnumeric	Determine if input is numeric array
isobject	Determine if input is MATLAB object
ispc	Determine if running MATLAB for PC (Windows) platform
isprime	Detect prime elements of array
isprop	Determine if input is object property
isreal	Determine if all array elements are real numbers
isscalar	Determine if input is scalar
issorted	Determine if set elements are in sorted order
isspace	Detect space characters in array
issparse	Determine if input is sparse array
isstrprop	Determine if string is of specified category
isstruct	Determine if input is MATLAB structure array
isstudent	Determine if Student Version of MATLAB

isunix	Determine if running MATLAB for UNIX[6] platform.
isvarname	Determine if input is valid variable name
isvector	Determine if input is vector

See Also

isa | exist

6. UNIX is a registered trademark of The Open Group in the United States and other countries.

Purpose

Determine if input is object of specified class

Syntax

```
tf = isa(obj,ClassName)
tf = isa(obj,classCategory)
```

Description

`tf = isa(obj,`*`ClassName`*`)` returns `true` if `obj` is an instance of the class specified by *`ClassName`*, and `false` otherwise. `isa` also returns true if `obj` is an instance of a class that is derived from *`ClassName`*.

`obj` can be any MATLAB variable.

`ClassName` can be any of the following:

- Name of any MATLAB class or fundamental type
- Name of a Java, or .NET class

MATLAB Fundamental Types

`'single'`	Single-precision number
`'double'`	Double-precision number
`'int8'`	Signed 8-bit integer
`'int16'`	Signed 16-bit integer
`'int32'`	Signed 32-bit integer
`'int64'`	Signed 64-bit integer
`'uint8'`	Unsigned 8-bit integer
`'uint16'`	Unsigned 16-bit integer
`'uint32'`	Unsigned 32-bit integer
`'uint64'`	Unsigned 64-bit integer
`'logical'`	Logical `true` or `false`
`'char'`	Character or string
`'struct'`	Structure array

`'cell'`	Cell array
`'function_handle'`	function handle

`tf = isa(obj,`*`classCategory`*`)` returns `true` if `obj` is an instance of any of the classes in the specified *`classCategory`*, and `false` otherwise. `isa` also returns true if `obj` is an instance of a class that is derived from any of the classes in *`classCategory`*.

`classCategory` can be `'numeric'`, `'float'`, or `'integer'`, representing a category of numeric types:

Categories of Numeric Types

`'numeric'`	Integer or floating-point array (`double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`)
`'float'`	Single- or double-precision floating-point array (`double`, `single`)
`'integer'`	Signed or unsigned integer array (`int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`)

To test for a sparse array, use `issparse`. To test for a complex array, use `~isreal`.

Examples

These examples show the values returned by `isa` when passed different types:

Determine if the value returned by the `pi` function is of class `double`:

```
isa(pi,'double')
ans =
     1
```

More generally, determine if the value returned by the `pi` function is a numeric value:

```
isa(pi,'numeric')
```

```
ans =
     1
```

isa also returns true for the float category because the class double is a floating-point type. However, pi does not return an integer type:

```
isa(pi,'integer')
ans =
     0
```

Determine if the 2–by-3 array returned by true is of type logical:

```
isa(true(2,3),'logical')
ans =
     1
```

Identify an instance of the MATLAB containers.Map class:

```
mapObj = containers.Map({'Color','RGB'},...
   {'Yellow',uint8([255,255,0])});
isa(mapObj,'containers.Map')
ans =
     1
```

The map key, RGB, references a uint8 array:

```
isa(mapObj('RGB'),'integer')
ans =
     1
```

Specifying a particular integer class provides more specific testing:

```
if strcmp(mapObj('Color'),'Yellow') && isa(mapObj('RGB'),'uint8')
   % The Color is Yellow and the RGB numbers are uint8 values
   ...
end
```

See Also

class | is* | isnumeric | isfloat | isinteger | exist

Purpose

True if application-defined data exists

Syntax

```
isappdata(h,name)
```

Description

`isappdata(h,name)` returns `1` if application-defined data with the specified `name` exists on the object specified by handle `h`, and returns `0` otherwise.

Tips

Application data is data that is meaningful to or defined by your application which you attach to a figure or any GUI component (other than ActiveX controls) through its `AppData` property. Only Handle Graphics MATLAB objects use this property.

See Also

`getappdata` | `rmappdata` | `setappdata`

iscell

Purpose

Determine whether input is cell array

Syntax

```
tf = iscell(A)
```

Description

`tf = iscell(A)` returns logical `1` (`true`) if `A` is a cell array and logical `0` (`false`) otherwise.

Examples

```
A{1,1} = [1 4 3; 0 5 8; 7 2 9];
A{1,2} = 'Anne Smith';
A{2,1} = 3+7i;
A{2,2} = -pi:pi/10:pi;

iscell(A)

ans =

     1
```

See Also

`cell` | `iscellstr` | `isstruct` | `isnumeric` | `islogical` | `isobject` | `isa` | `is*`

Purpose

Determine whether input is cell array of strings

Syntax

```
tf = iscellstr(A)
```

Description

`tf = iscellstr(A)` returns logical `1` (`true`) if `A` is a cell array of strings (or an empty cell array), and logical `0` (`false`) otherwise. A cell array of strings is a cell array where every element is a character array.

Examples

```
A{1,1} = 'Thomas Lee';
A{1,2} = 'Marketing';
A{2,1} = 'Allison Jones';
A{2,2} = 'Development';

iscellstr(A)

ans =

     1
```

See Also

`cellstr` | `iscell` | `isstrprop` | `strings` | `char` | `isstruct` | `isa` | `is*`

ischar

Purpose

Determine whether item is character array

Syntax

```
tf = ischar(A)
```

Description

`tf = ischar(A)` returns logical `1` (`true`) if `A` is a character array and logical `0` (`false`) otherwise.

Examples

Given the following cell array,

```
C{1,1} = magic(3);            % double array
C{1,2} = 'John Doe';          % char array
C{1,3} = 2 + 4i               % complex double

C =

    [3x3 double]    'John Doe'    [2.0000+ 4.0000i]
```

`ischar` shows that only `C{1,2}` is a character array.

```
for k = 1:3
x(k) = ischar(C{1,k});
end

x

x =

     0     1     0
```

See Also

`char` | `strings` | `isletter` | `isspace` | `isstrprop` | `iscellstr` | `isnumeric` | `isa` | `is*`

Purpose

Determine whether input is column vector

Syntax

```
iscolumn(V)
```

Description

`iscolumn(V)` returns logical 1 (`true`) if `size(V)` returns `[n 1]` with a nonnegative integer value `n`, and logical 0 (`false`) otherwise.

Examples

Determine if a vector is a column. This example is a row so `iscolumn` returns 0:

```
V = rand(1,5);
iscolumn(V)
ans =
     0
```

Transpose the vector to make it a column. `iscolumn` returns 1:

```
V1 = V';
iscolumn(V1)
ans =
     1
```

See Also

`ismatrix` | `isrow` | `isscalar` | `isvector`

Purpose

Determine whether input is COM or ActiveX object

Syntax

```
tf = h.iscom
tf = iscom(h)
```

Description

`tf = h.iscom` returns logical `1` (`true`) if handle `h` is a COM or Microsoft ActiveX object. Otherwise, returns logical `0` (`false`).

`tf = iscom(h)` is an alternate syntax.

COM functions are available on Microsoft Windows systems only.

Examples

Test an instance of a Microsoft Excel application:

```
h = actxserver('Excel.Application');
h.iscom
```

MATLAB displays `true`, indicating object `h` is a COM object.

Test an Excel interface:

```
h = actxserver('Excel.Application');
%Create a workbooks object
w = h.get('workbooks');
w.iscom
```

MATLAB displays `false`, indicating object `w` is not a COM object.

How To

- “MATLAB COM Integration”

Purpose

Determine whether input is folder

Syntax

```
tf = isdir('A')
```

Description

`tf = isdir('A')` returns logical `1` (`true`) if `A` is a folder. Otherwise, it returns logical `0` (`false`).

Examples

Run:

```
tf=isdir('myfiles/results')
```

MATLAB returns

```
tf =
 1
```

indicating that `myfiles/results` is a folder.

See Also

`dir` | `is*`

TriRep.isEdge

Purpose

(Will be removed) Test if vertices are joined by edge

Note `isEdge(TriRep)` will be removed in a future release. Use `isConnected(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
TF = isEdge(TR, V1, V2)
TF = isEdge(TR, EDGE)
```

Description

`TF = isEdge(TR, V1, V2)` returns an array of 1/0 (true/false) flags, where each entry `TF(i)` is true if `V1(i)`, `V2(i)` is an edge in the triangulation. `V1`, `V2` are column vectors representing the indices of the vertices in the mesh, that is, indices into the vertex coordinate arrays.

`TF = isEdge(TR, EDGE)` specifies the edge start and end indices in matrix format.

Input Arguments

`TR`	Triangulation representation.
`V1, V2`	Column vectors of mesh vertices.
`EDGE`	Matrix of size `n`-by-2 where `n` is the number of query edges.

Output Arguments

`TF`	Array of 1/0 (true/false) flags, where each entry `TF(i)` is true if `V1(i)`, `V2(i)` is an edge in the triangulation.

Examples

Example 1

Load a 2-D triangulation and use `TriRep` to query the presence of an edge between pairs of points.

```
load trimesh2d
trep = TriRep(tri, x,y);
```

Test if vertices 3 and 117 are connected by an edge

```
isEdge(trep, 3, 117)
```

Test if vertices 3 and 164 are connected by an edge

```
isEdge(trep, 3, 164)
```

Example 2

Direct query of a 3-D Delaunay triangulation created using `DelaunayTri`.

```
X = rand(10,3)
dt = DelaunayTri(X)
```

Test if vertices 2 and 7 are connected by an edge

```
isEdge(dt, 2, 7);
```

See Also

`delaunayTriangulation` | `triangulation`

isempty

Purpose

Determine whether array is empty

Syntax

```
TF = isempty(A)
```

Description

`TF = isempty(A)` returns logical 1 (`true`) if `A` is an empty array and logical 0 (`false`) otherwise. An empty array has at least one dimension of size zero, for example, 0-by-0 or 0-by-5.

Examples

```
B = rand(2,2,2);
B(:,:,:) = [];

isempty(B)

ans = 1
```

See Also

`is*`

isempty (tscollection)

Purpose Determine whether `tscollection` object is empty

Syntax `isempty(tsc)`

Description `isempty(tsc)` returns a logical value for `tscollection` object `tsc`, as follows:

- `1` — When `tsc` contains neither `timeseries` members nor a time vector
- `0` — When `tsc` contains either `timeseries` members or a time vector

See Also `timeseries` | `tscollection` | `length (tscollection)` | `size (tscollection)`

isequal

Purpose

Test arrays for equality

Syntax

```
tf = isequal(A, B, ...)
```

Description

`tf = isequal(A, B, ...)` returns logical `1` (`true`) if the input arrays have the same contents, and logical `0` (`false`) otherwise. Nonempty arrays must be of the same data type and size.

Tips

When comparing structures, the order in which the fields of the structures were created is not important. As long as the structures contain the same fields, with corresponding fields set to equal values, `isequal` considers the structures to be equal. See Example 2, below.

When comparing numeric values, `isequal` does not consider the data type used to store the values in determining whether they are equal. See Example 3, below. This is also true when comparing numeric values with certain nonnumeric values, such as logical `true` and 1, or the character `A` and its numeric equivalent, 65.

When comparing handle objects, use `eq` or the == operator to test whether objects are the same handle. Use `isequal` to test if objects have equal property values, even if those objects are different handles.

`NaN`s (Not a Number), by definition, are not equal. Therefore, arrays that contain `NaN` elements are not equal, and `isequal` returns zero when comparing such arrays. See Example 4, below. Use the `isequaln` function when you want to test for equality with `NaN`s treated as equal.

`isequal` recursively compares the contents of cell arrays and structures. If all the elements of a cell array or structure are numerically equal, `isequal` returns logical `1`.

Examples

Example 1

Given

```
A =                      B =                      C =
      1     0                  1     0                  1     0
      0     1                  0     1                  0     0
```

isequal(A,B,C) returns 0, and isequal(A,B) returns 1.

Example 2

When comparing structures with isequal, the order in which the fields of the structures were created is not important:

```
A.f1 = 25;    A.f2 = 50
A =
    f1: 25
    f2: 50

B.f2 = 50;    B.f1 = 25
B =
    f2: 50
    f1: 25

isequal(A, B)
ans =
     1
```

Example 3

When comparing numeric values, the data types used to store the values are not important:

```
A = [25 50];   B = [int8(25) int8(50)];

isequal(A, B)
ans =
     1
```

Example 4

Arrays that contain NaN (Not a Number) elements cannot be equal, since NaNs, by definition, are not equal:

```
A = [32 8 -29 NaN 0 5.7];
B = A;
```

```
isequal(A, B)
ans =
     0
```

See Also

isequaln | eq | strcmp | isa | is* | relational operators

Purpose

Test arrays for equality, treating NaN values as equal

Syntax

```
tf = isequaln(A,B)
tf = isequaln(A1,A2,...,An)
```

Description

`tf = isequaln(A,B)` returns `true` if `A` and `B` are the same size and their contents are of equal value. Otherwise, `isequaln` returns `false`. All `NaN` (not a number) values are considered to be equal to each other. `isequaln` recursively compares the contents of cell arrays and structures and the properties of objects. If all contents in the respective locations are numerically equal, `isequaln` returns `true`.

`tf = isequaln(A1,A2,...,An)` returns logical `true` if all the inputs are numerically equal.

Input Arguments

A,B - Inputs to be compared

arrays | structures | cell arrays | objects

Inputs to be compared, specified as arrays, structures, cell arrays, or objects containing numeric or char values. The numeric types of `A` and `B` do not have to match. If `A` and `B` are structures, their fields need not be in the same order.

`isequaln` returns `false` when you pass objects of different classes. This applies even when the objects have the same properties and their values match.

Data Types

`double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `cell` | `struct`

Complex Number Support: Yes

A1,A2,...,An - Series of inputs to be compared

arrays | structures | cell arrays | objects

Series of inputs to be compared, specified as arrays, structures, cell arrays, or objects containing numeric or char values. The numeric types

of A1,A2,...,An do not have to match. If A1,A2,...,An are structures, their fields need not be in the same order.

isequaln returns false when you pass objects of different classes. This applies even when the objects have the same properties and their values match.

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

Output Arguments

tf - Result

true | false

Result, returned as a logical value. isequaln returns true if the inputs are numerically equal. Otherwise, it returns false.

Tips

- If you want to test handle objects for equality, use eq or the == operator to test whether they are the same handle. Use isequaln to test handle objects for equal property values, even when the objects are different handles.
- Use isequal if you want to test for equality and treat NaN values as unequal.

Examples

Compare Two Arrays

```
A = zeros(3,3)+1e-20;
B = zeros(3,3);
tf = isequaln(A,B)
```

```
tf =

     0
```

Compare Two Structures

```
A = struct('field1',0.005,'field2',2500);
```

```
B = struct('field2',2500,'field1',0.005);
tf = isequaln(A,B)

tf =

     1
```

Even though the ordering of the fields in each structure is different, `isequaln` treats them as the same because the values are equal.

Comparing Numeric Values with Special Nonnumeric Values

Compare the logical value `true` to the double integer `1`.

```
isequaln(true,1)

ans =

     1
```

Notice that `isequaln` does not consider data type when it tests for equality.

Compare `'A'` to the ASCII-equivalent integer, `65`.

```
isequaln('A',65)

ans =

     1
```

Compare Vectors Containing NaN Values

```
A1 = [1 NaN NaN];
A2 = [1 NaN NaN];
A3 = [1 NaN NaN];
tf = isequaln(A1,A2,A3)

tf =
```

1

See Also

isequal | eq | is* | isa* | strcmp | relational operators

Purpose

Compare scalar `MException` objects for equality

Syntax

`TF = isequal(eObj1, eObj2)`

Description

`TF = isequal(eObj1, eObj2)` tests `MException` objects `eObj1` and `eObj2` for equality, returning logical 1 (`true`) if the two objects are identical, otherwise returning logical 0 (`false`).

See Also

`try` | `catch` | `error` | `assert` | `MException` | `eq(MException)` | `ne(MException)` | `getReport(MException)` | `disp(MException)` | `throw(MException)` | `rethrow(MException)` | `throwAsCaller(MException)` | `addCause(MException)` | `last(MException)`

isequalwithequalnans

Purpose

Test arrays for equality, treating NaNs as equal

Note `isequalwithequalnans` is not recommended. Use `isequaln` instead.

Syntax

```
tf = isequalwithequalnans(A, B, ...)
```

Description

`tf = isequalwithequalnans(A, B, ...)` returns logical `1` (`true`) if the input arrays are the same type and size and hold the same contents, and logical `0` (`false`) otherwise. `NaN` (Not a Number) values are considered to be equal to each other. Numeric data types and structure field order do not have to match.

Tips

`isequalwithequalnans` is the same as `isequal`, except `isequalwithequalnans` considers `NaN` (Not a Number) values to be equal, and `isequal` does not.

`isequalwithequalnans` recursively compares the contents of cell arrays and structures. If all the elements of a cell array or structure are numerically equal, `isequalwithequalnans` returns logical `1`.

Examples

Arrays containing `NaN`s are handled differently by `isequal` and `isequalwithequalnans`. `isequal` does not consider `NaN`s to be equal, while `isequalwithequalnans` does.

```
A = [32 8 -29 NaN 0 5.7];
B = A;
isequal(A, B)
ans =
     0

isequalwithequalnans(A, B)
ans =
     1
```

The position of NaN elements in the array does matter. If they are not in the same position in the arrays being compared, then isequalwithequalnans returns zero.

```
A = [2 4 6 NaN 8];   B = [2 4 NaN 6 8];
isequalwithequalnans(A, B)
ans =
     0
```

See Also

isequal | strcmp | isa | is* | relational operators | isequaln

isevent

Purpose

Determine whether input is COM object event

Syntax

```
tf = h.isevent('eventname')
tf = isevent(h, 'eventname')
```

Description

`tf = h.isevent('eventname')` returns logical `1` (`true`) if `event_name` is an event recognized by COM object `h`. Otherwise, returns logical `0` (`false`). The `event_name` argument is not case sensitive.

`tf = isevent(h, 'eventname')` is an alternate syntax.

COM functions are available on Microsoft Windows systems only.

Examples

Test events in a MATLAB sample control object:

1. Create an instance of the `mwsamp` control and test `DblClick`:

   ```
   f = figure ('position', [100 200 200 200]);
   h = actxcontrol ('mwsamp.mwsampctrl.2', [0 0 200 200], f);
   h.isevent('DblClick')
   ```

   MATLAB displays `true`, indicating `DblClick` is an event.

2. Try the same test on`Redraw`:

   ```
   h.isevent('Redraw')
   ```

   MATLAB displays `false`, indicating `Redraw` is not an event; it is a method.

Test events in a Microsoft Excel workbook object:

1. Create a `Workbook` object `wb`:

   ```
   myApp = actxserver('Excel.Application');
   wbs = myApp.Workbooks;
   wb = wbs.Add;
   ```

2 Test `Activate`:

```
wb.isevent('Activate')
```

MATLAB displays `true`, indicating `Activate` is an event.

3 Test `Save`:

```
wb.isevent('Save')
```

MATLAB displays `false`, indicating `Save` is not an event; it is a method.

See Also

`events (COM)` | `eventlisteners` | `registerevent`

How To

- "Exploring Events"
- "Functions for Working with Events"

Purpose Determine whether input is structure array field

Syntax

```
tf = isfield(S, 'fieldname')
tf = isfield(S, C)
```

Description

`tf = isfield(S, 'fieldname')` examines structure `S` to see if it includes the field specified by the quoted string `'fieldname'`. Output `tf` is set to logical `1` (`true`) if `S` contains the field, or logical `0` (`false`) if not. If `S` is not a structure array, `isfield` returns `false`.

`tf = isfield(S, C)` examines structure `S` for multiple fieldnames as specified in cell array of strings `C`, and returns an array of logical values to indicate which of these fields are part of the structure. Elements of output array `tf` are set to a logical 1 (`true`) if the corresponding element of `C` holds a fieldname that belongs to structure `S`. Otherwise, logical 0 (`false`) is returned in that element. In other words, if structure `S` contains the field specified in `C{m,n}`, `isfield` returns a logical 1 (`true`) in `tf(m,n)`.

Note `isfield` returns `false` if the `field` or `fieldnames` input is empty.

Examples

Example 1 — Single Fieldname Syntax

Given the following MATLAB structure,

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

`isfield` identifies `billing` as a field of that structure.

```
isfield(patient,'billing')
ans =
     1
```

Example 2 — Multiple Fieldname Syntax

Check structure `S` for any of four possible fieldnames. Only the first is found, so the first element of the return value is set to `true`:

```
S = struct('one', 1, 'two', 2);

fields = isfield(S, {'two', 'pi', 'One', 3.14})
fields =
     1     0     0     0
```

See Also

`fieldnames` | `setfield` | `getfield` | `orderfields` | `rmfield` | `struct` | `isstruct` | `iscell` | `isa` | `is*`

How To

- dynamic field names

isfinite

Purpose

Array elements that are finite

Syntax

```
TF = isfinite(A)
```

Description

`TF = isfinite(A)` returns an array the same size as `A` containing logical 1 (`true`) where the elements of the array `A` are finite and logical 0 (`false`) where they are infinite or `NaN`. For a complex number `z`, `isfinite(z)` returns `1` if both the real and imaginary parts of `z` are finite, and `0` if either the real or the imaginary part is infinite or `NaN`.

For any real `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, and `isnan(A)` is equal to one.

Examples

```
a = [-2  -1  0  1  2];
isfinite(1./a)
ans =
     1     1     0     1     1
isfinite(0./a)
ans =
     1     1     0     1     1
```

See Also

`isinf` | `isnan` | `is*`

Purpose

Determine if input is floating-point array

Syntax

```
tf = isfloat(A)
```

Description

`tf = isfloat(A)` returns `true` if `A` is a floating-point array and `false` otherwise. The floating-point types are `single` and `double`, and subclasses of `single` and `double`.

Examples

These examples show the values `isfloat` returns when passed specific types:

```
% pi returns a floating-point value
isfloat(pi)
ans =
     1
% Complex numbers are floating-point values
isfloat(3+7i)
ans =
     1
% Single-precision numbers are floating-point values
isfloat(realmax('single'))
ans =
     1
% isfloat returns a logical value
isfloat(isfloat(pi))
ans =
     0
```

See Also

`isa` | `isinteger` | `double` | `single` | `isnumeric`

isglobal

Purpose

Determine whether input is global variable

Note Support for the `isglobal` function will be removed in a future release of the MATLAB software. See "Tips" on page 1-2846 below.

Syntax

```
tf = isglobal(A)
```

Description

`tf = isglobal(A)` returns logical `1` (`true`) if `A` has been declared to be a global variable in the context from which `isglobal` is called, and logical `0` (`false`) otherwise.

Tips

`isglobal` is most commonly used in conjunction with conditional global declaration. An alternate approach is to use a pair of variables, one local and one declared global.

Instead of using

```
if condition
   global x
end

x = some_value

if isglobal(x)
   do_something
end
```

You can use

```
global gx
if condition
   gx = some_value
else
   x = some_value
end
```

```
if condition
   do_something
end
```

If no other workaround is possible, you can replace the command

```
isglobal(variable)
```

with

```
~isempty(whos('global','variable'))
```

See Also

global | isvarname | isa | is*

ishandle

Purpose

Test for valid object handle

Syntax

```
ishandle(H)
```

Description

`ishandle(H)` returns an array whose elements are `1` where the elements of `H` are object handles, and `0` where they are not.

Note You should use the most specific function for your application, as described in the following sections.

MATLAB Object Handles

Use the `isa` function to determine the class of MATLAB objects and use the `isvalid handle` class method to determine the validity of handle objects. See "Testing Handle Validity" for information on testing for MATLAB handle objects.

Graphics Object Handles

Use `ishghandle` for graphics objects.

Java Object Handles

Use `isjava` for Java objects.

See Also

`isa` | `isjava` | `ishghandle`

Purpose

True for Handle Graphics object handles

Syntax

```
ishghandle(h)
```

Description

`ishghandle(h)` returns an array that contains 1's where the elements of `h` are handles to existing graphic objects and 0's where they are not. Differs from `ishandle` in that Simulink objects handles return false.

Examples

Create a plot and find the valid handles:

```
x = [1:10];
y = [1:10];
p=plot(x,y);
ishghandle([x y p])
% This returns a 1-by-21 array of values with ones at the first,
% eleventh, and last values, if the figure handle is 1.
```

See Also

`isa` | `ishandle` | `findobj` | `gca` | `gcf` | `set`

How To

- "Accessing Object Handles"

ishold

Purpose

Current hold state

Syntax

```
ishold
```

Description

`ishold` returns `1` if `hold` is `on`, and `0` if it is `off`. When `hold` is `on`, the current plot and most axis properties are held so that subsequent graphing commands add to the existing graph.

A state of `hold on` implies that both figure and axes `NextPlot` properties are set to `add`.

See Also

`hold` | `newplot`

How To

- "Controlling Graphics Output"

Purpose

Array elements that are infinite

Syntax

```
TF = isinf(A)
```

Description

`TF = isinf(A)` returns an array the same size as `A` containing logical 1 (`true`) where the elements of `A` are `+Inf` or `-Inf` and logical 0 (`false`) where they are not. For a complex number `z`, `isinf(z)` returns `1` if either the real or imaginary part of `z` is infinite, and `0` if both the real and imaginary parts are finite or `NaN`.

For any real `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, and `isnan(A)` is equal to one.

Examples

```
a = [-2  -1  0  1  2];

isinf(1./a)

ans =
     0     0     1     0     0

isinf(0./a)

ans =
     0     0     0     0     0
```

See Also

`isfinite` | `isnan` | `is*`

isinteger

Purpose

Determine if input is integer array

Syntax

```
tf = isinteger(A)
```

Description

`tf = isinteger(A)` returns `true` if the array `A` is an integer type and `false` otherwise.

An integer array is any of the following integer types and any subclasses of those types:

MATLAB Integer Types

`int8`	8-bit signed integer array
`uint8`	8-bit unsigned integer array
`int16`	16-bit signed integer array
`uint16`	16-bit unsigned integer array
`int32`	32-bit signed integer array
`uint32`	32-bit unsigned integer array
`int64`	64-bit signed integer array
`uint64`	64-bit unsigned integer array

Examples

These examples show the values `isinteger` returns when passed specific types:

```
% uint8 is one of the integer types
isinteger(uint8(1:255))
ans =
     1
% pi returns a double-precision value
isinteger(pi)
ans =
     0
% Constants are double-precision by default
isinteger(3)
```

```
ans =
     0
% isinteger returns a logical value
isinteger(isinteger(uint8(3)))
ans =
     0
```

See Also

isa | isnumeric | isfloat

isinterface

Purpose

Determine whether input is COM interface

Syntax

```
tf = h.isinterface
tf = isinterface(h)
```

Description

`tf = h.isinterface` returns logical `1` (`true`) if handle `h` is a COM interface. Otherwise, returns logical `0` (`false`).

`tf = isinterface(h)` is an alternate syntax.

COM functions are available on Microsoft Windows systems only.

Examples

Test an instance of a Microsoft Excel application:

```
h = actxserver('Excel.Application');
h.isinterface
```

MATLAB displays `false`, indicating object `h` is not an interface.

Test a `workbooks` object:

```
w = h.get('workbooks');
w.isinterface
```

MATLAB displays `true`, indicating object `w` is an interface.

See Also

`iscom` | `interfaces`

How To

- "Exploring Interfaces"

Purpose

Determine if input is Java object

Syntax

```
tf = isjava(A)
```

Description

`tf = isjava(A)` returns logical `1` (`true`) if object `A` is a Java object. Otherwise, it returns logical `0` (`false`).

Examples

Test if java.util.Date Is Java Object

Create an instance of the Java Date class.

```
myDate = java.util.Date;
isjava(myDate)
```

```
ans =

     1
```

`myDate` is a Java object.

However, `myDate` is not a MATLAB object.

```
isobject(myDate)
```

```
ans =

     0
```

See Also

`isobject` | `javaArray` | `javaMethod` | `javaObject` | `isa` | `is*`

containers.Map.isKey

Purpose

Determine if `containers.Map` object contains key

Syntax

```
tf = isKey(mapObj,keySet)
```

Description

`tf = isKey(mapObj,keySet)` looks for the specified keys in `mapObj`, and returns logical `true` (1) for the keys that it finds, and logical `false` (0) for those it does not. `keySet` is a scalar key or a cell array of keys.

Input Arguments

mapObj

Object of class `containers.Map`.

keySet

Scalar value, string, or cell array that specifies keys to find in `mapObj`.

Output Arguments

tf

Array of logical values. If `keySet` is a scalar or a string, `tf` is a scalar. Otherwise, `tf` has the same size and dimensions as `keySet`.

Examples

Find Keys in a Map

Construct a map that contains rainfall data for several months:

```
months =   {'Jan', 'Feb', 'Mar', 'Apr'};
rainfall = [327.2, 368.2, 197.6, 178.4];
mapObj = containers.Map(months,rainfall);
```

Determine if keys `Apr`, `May`, and `Jun` are in the map:

```
keySet = {'Apr','May','Jun'};
tf = isKey(mapObj,keySet)
```

This code returns 1-by-3 vector `tf`:

```
tf =
     1     0     0
```

Find a Single Key

Determine if `mapObj` from the previous example contains key `Feb`:

```
keySet = 'Feb';
tf = isKey(mapObj,keySet)
```

This code returns scalar `tf`:

```
tf =
     1
```

See Also

`containers.Map` | `keys` | `values` | `remove`

iskeyword

Purpose

Determine whether input is MATLAB keyword

Syntax

```
tf = iskeyword('str')
iskeyword str
iskeyword
```

Description

`tf = iskeyword('str')` returns logical `1` (`true`) if the string `str` is a keyword in the MATLAB language and logical `0` (`false`) otherwise.

`iskeyword str` uses the MATLAB command format.

`iskeyword` returns a list of all MATLAB keywords.

Examples

To test if the word `while` is a MATLAB keyword,

```
iskeyword while
ans =
     1
```

To obtain a list of all MATLAB keywords,

```
iskeyword
    'break'
    'case'
    'catch'
    'classdef'
    'continue'
    'else'
    'elseif'
    'end'
    'for'
    'function'
    'global'
    'if'
    'otherwise'
    'parfor'
    'persistent'
    'return'
```

```
'spmd'
'switch'
'try'
'while'
```

See Also

isvarname | genvarname | is*

isletter

Purpose

Array elements that are alphabetic letters

Syntax

```
tf = isletter('str')
```

Description

`tf = isletter('str')` returns an array the same size as `str` containing logical `1` (`true`) where the elements of `str` are letters of the alphabet and logical `0` (`false`) where they are not.

Examples

Find the letters in character array `s`.

```
s = 'A1,B2,C3';

isletter(s)
ans =
     1     0     0     1     0     0     1     0
```

See Also

`ischar` | `isspace` | `isstrprop` | `iscellstr` | `isnumeric` | `char` | `strings` | `isa` | `is*`

Purpose

Determine if input is logical array

Syntax

```
tf = islogical(A)
```

Description

`tf = islogical(A)` returns `true` if `A` is a logical array and `false` otherwise. `islogical` also returns `true` if `A` is an instance of a class that is derived from the `logical` class.

Examples

These examples show the values `islogical` returns when passed specific types:

```
% Relational operators return logical values
islogical(5<7)
ans =
     1

% true and false return logical values
islogical(true) & islogal(false)
ans =
     1

% Constants are double-precision by default
islogical(1)
ans =
     0

% logical creates logical values
islogical(logical(1))
ans =
     1
```

See Also

`logical` | `elementwise` | `short-circuit` | `isa` | `is*`

ismac

Purpose

Determine if version is for Mac OS X platform

Syntax

```
tf = ismac
```

Description

`tf = ismac` returns logical `1` (`true`) if the version of MATLAB software is for the Apple Mac OS X platform, and returns logical `0` (`false`) otherwise.

Tips

- The `isunix` function also determines if version is for Mac OS X platforms.

See Also

`isunix` | `ispc` | `computer` | `isstudent` | `is*`

Purpose

Determine whether input is matrix

Syntax

```
ismatrix(V)
```

Description

`ismatrix(V)` returns logical 1 (`true`) if `size(V)` returns `[m n]` with nonnegative integer values `m` and `n`, and logical 0 (`false`) otherwise.

Examples

Create three vectors:

```
V1 = rand(5,1);
V2 = rand(5,1);
V3 = rand(5,1);
```

Concatenate the vectors and check that the result is a matrix. `ismatrix` returns 1:

```
M = cat(2,V1,V2,V3);
ismatrix(M)
ans =
     1
```

See Also

`iscolumn` | `isrow` | `isscalar` | `isvector`

ismember

Purpose

Array elements that are members of set array

Syntax

```
Lia = ismember(A,B)
Lia = ismember(A,B,'rows')
[Lia,Locb] = ismember(A,B)
[Lia,Locb] = ismember(A,B,'rows')

[Lia,Locb] = ismember(___,'legacy')
```

Description

`Lia = ismember(A,B)` returns an array the same size as `A`, containing `1` (`true`) where the elements of `A` are found in `B`. Elsewhere, it returns `0` (`false`).

`Lia = ismember(A,B,'rows')` treats each row of `A` and each row of `B` as single entities and returns a vector containing `1` (`true`) where the rows of matrix `A` are also rows of `B`. Elsewhere, it returns `0` (`false`).

`A` and `B` must have the same number of columns when you use the `'rows'` option. Furthermore, the `'rows'` option does not support cell arrays.

`[Lia,Locb] = ismember(A,B)` also returns an array, `Locb`, containing the lowest index in `B` for each value in `A` that is a member of `B`. The output array, `Locb`, contains `0` wherever `A` is not a member of `B`.

`[Lia,Locb] = ismember(A,B,'rows')` also returns a vector, `Locb`, containing the lowest index in `B` for each row in `A` that is also a row in `B`. The output vector, `Locb`, contains `0` wherever `A` is not a row of `B`.

`[Lia,Locb] = ismember(___,'legacy')` preserves the behavior of the `ismember` function from R2012b and prior releases using any of the input arguments in previous syntaxes.

Input Arguments

A - Query array

vector | matrix | N-D array

Query array, specified as a vector, matrix, or N-D array, contains the values to search for in the set array, B. Elements of the query array, A, can be any numeric, logical, or char data type. A also can be a cell array of strings. Furthermore, A can be an object with the following class methods:

- sort (or sortrows for the 'rows' option), where the second output of this method is double or another built-in numeric class
- eq
- ne

These objects include heterogeneous arrays derived from the same root class.

A must belong to the same class as B with the following exceptions:

- logical, char, and all numeric classes can combine with double arrays.
- Cell arrays of strings can combine with char arrays.

If you specify the 'rows' option, A must have the same number of columns as B.

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

B - Set array

vector | matrix | N-D array

Set array, specified as a vector, matrix, or N-D array, contains the values that evaluate to true wherever they are found in the query array, A. Elements of the set array, B, can be any numeric, logical, or char data type. B also can be a cell array of strings. Furthermore, B can be an object with the following class methods:

- sort (or sortrows for the 'rows' option), where the second output of this method is double or another built-in numeric class
- eq
- ne

These objects include heterogeneous arrays derived from the same root class.

B must belong to the same class as A with the following exceptions:

- logical, char, and all numeric classes can combine with double arrays.
- Cell arrays of strings can combine with char arrays.

If you specify the 'rows' option, B must have the same number of columns as A.

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

Output Arguments

Lia - Logical index to A
vector | matrix | N-D array

Logical index to A, returned as a vector, matrix or N-D array containing 1 (true) wherever the values (or rows) in A are members of B. Elsewhere, it returns 0 (false).

Lia is an array of the same size as A, unless you specify the 'rows' flag.

If the 'rows' flag is specified, Lia is a column vector with the same number of rows as A.

Locb - Locations in B
vector | matrix | N-D array

Locations in B, returned as a vector, matrix, or N-D array. If the 'legacy' flag is not specified, Locb contains the lowest indices to the

values (or rows) in `B` that are found in `A`. `Locb` contains `0` wherever `A` is not a member of `B`.

`Locb` is an array of the same size as `A` unless you specify the `'rows'` flag.

If the `'rows'` flag is specified, `Locb` is a column vector with the same number of rows as `A`.

Examples

Values That Are Members of a Set

Define two vectors with values in common.

```
A = [5 3 4 2]; B = [2 4 4 4 6 8];
```

Determine which elements of `A` are also in `B`.

```
Lia = ismember(A,B)
```

```
Lia =

     0     0     1     1
```

`A(3)` and `A(4)` are found in `B`.

Members of a Set and Indices to Set Values

Define two vectors with values in common.

```
A = [5 3 4 2]; B = [2 4 4 4 6 8];
```

Determine which elements of `A` are also in `B` as well as their corresponding locations in `B`.

```
[Lia,Locb] = ismember(A,B)
```

```
Lia =

     0     0     1     1

Locb =
```

```
     0     0     2     1
```

The lowest index to `A(3)` is `B(2)`.

`A(4)` is found in `B(1)`.

Rows That Belong to a Set

Define two matrices with a row in common.

```
A = [1 3 5 6; 2 4 6 8];
B = [2 4 6 8; 1 3 5 7; 2 4 6 8];
```

Determine which rows of `A` are also in `B` as well as their corresponding locations in `B`.

```
[Lia, Locb] = ismember(A,B, 'rows')

Lia =

     0
     1


Locb =

     0
     1
```

The lowest index to `A(2,:)` is `B(1,:)`.

Members of Set Containing NaN Values

Define two vectors containing `NaN`.

```
A = [5 NaN NaN]; B = [5 NaN NaN];
```

Determine which elements of `A` are also in `B` as well as their corresponding locations in `B`.

```
[Lia,Locb] = ismember(A,B)

Lia =

     1     0     0


Locb =

     1     0     0
```

`ismember` treats `NaN` values as distinct.

Cell Array of Strings with Trailing White Space

Create a cell array of strings, `A`.

```
A = {'dog','cat','fish','horse'};
```

Create a cell array of strings, `B`, where some of the strings have trailing white space.

```
B = {'dog ','cat','fish ','horse'};
```

Determine which strings of `A` can be found in `B`.

```
[Lia,Locb] = ismember(A,B)

Lia =

     0     1     0     1


Locb =

     0     2     0     4
```

`ismember` treats trailing white space in cell arrays of strings as distinct characters.

Members of Char and Cell Array of Strings

Create a character array, A.

```
A = ['cat';'dog';'fox';'pig'];
```

Create a cell array of strings, B.

```
B = {'dog','cat','fish','horse'};
```

Determine which strings of A can be found in B.

```
[Lia,Locb] = ismember(A,B)
```

```
Lia =

     1
     1
     0
     0


Locb =

     2
     1
     0
     0
```

Preserve Legacy Behavior of ismember

Use the 'legacy' flag to preserve the behavior of ismember from R2012b and prior releases in your code.

Find the members of B with the current behavior.

```
A = [5 3 4 2]; B = [2 4 4 4 6 8];
[Lia1,Locb1] = ismember(A,B)
```

```
Lia1 =

     0     0     1     1

Locb1 =

     0     0     2     1
```

Find the members of `B`, and preserve the legacy behavior.

```
[Lia2,Locb2] = ismember(A,B,'legacy')

Lia2 =

     0     0     1     1

Locb2 =

     0     0     4     1
```

See Also

`unique` | `intersect` | `union` | `issorted` | `setdiff` | `setxor` | `sort`

ismethod

Purpose

Determine if method of object

Syntax

```
tf = ismethod(h, 'methodName')
```

Description

`tf = ismethod(h, 'methodName')` returns logical `1` (`true`) if the specified `methodName` is a public method of object `obj`. Otherwise, returns logical `0` (`false`).

Examples

Determine if objects support equality testing:

```
if ismethod(obj1,'eq') && ismethod(obj2,'eq')
   tf = obj1 == obj2;
end
```

See Also

`methods` | `isprop` | | `isobject` | `class`

Tutorials

- "Methods"

ismethod (COM)

Purpose

Determine whether input is COM object method

Syntax

```
tf = h.ismethod('methodname')
tf = ismethod(h,'methodname')
```

Description

`tf = h.ismethod('methodname')` returns logical `1` (`true`) if the specified `methodname` is a method you can call on COM object `h`. Otherwise, returns logical `0` (`false`).

`tf = ismethod(h,'methodname')` is an alternate syntax.

COM functions are available on Microsoft Windows systems only.

Examples

Test members of an instance of a Microsoft Excel application:

```
h = actxserver ('Excel.Application');
ismethod(h,'SaveWorkspace')
```

MATLAB returns `true`, `SaveWorkspace` is a method.

Try the same test on `UsableWidth`:

```
ismethod(h,'UsableWidth')
```

MATLAB returns `false`, `UsableWidth` is not a method; it is a property.

See Also

`methods` | `methodsview` | `isprop` | `isevent` | `isobject` | `class`

How To

- “Exploring Methods”

isnan

Purpose

Array elements that are NaN

Syntax

```
TF = isnan(A)
```

Description

`TF = isnan(A)` returns an array the same size as `A` containing logical 1 (`true`) where the elements of `A` are `NaN`s and logical 0 (`false`) where they are not. For a complex number `z`, `isnan(z)` returns `1` if either the real or imaginary part of `z` is `NaN`, and `0` if both the real and imaginary parts are finite or `Inf`.

For any real `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, and `isnan(A)` is equal to one.

Examples

```
A = [-2  -1  0  1  2];

isnan(1./A)
ans =
     0     0     0     0     0

isnan(0./A)
ans =
     0     0     1     0     0
```

See Also

`isfinite` | `isinf` | `is*`

Purpose

Determine if input is numeric array

Syntax

```
tf = isnumeric(A)
```

Description

`tf = isnumeric(A)` returns `true` if `A` is a numeric array and `false` otherwise.

A numeric array is any of the following numeric types and any subclasses of those types:

MATLAB Numeric Types

`single`	Single-precision floating-point array
`double`	Double-precision floating-point array
`int8`	8-bit signed integer array
`uint8`	8-bit unsigned integer array
`int16`	16-bit signed integer array
`uint16`	16-bit unsigned integer array
`int32`	32-bit signed integer array
`uint32`	32-bit unsigned integer array
`int64`	64-bit signed integer array
`uint64`	64-bit unsigned integer array

Examples

These examples show the values `isnumeric` returns when passed specific types:

```
% pi returns a numeric value
isnumeric(pi)
ans =
     1
% Complex numbers are numeric
isnumeric(3+7i)
ans =
```

```
     1
% Integers are numeric
isnumeric(uint8(1:255))
ans =
     1
% isnumeric returns a logical value
isnumeric(isnumeric(pi))
ans =
     0
```

See Also

isfloat | isinteger | isnan | isreal | isprime | isfinite | isinf | isa | is*

Purpose Determine if input is MATLAB object

Syntax

```
tf = isobject(A)
```

Description

`tf = isobject(A)` returns `true` if `A` is an object of a MATLAB class. Otherwise, it returns `false`.

Handle Graphics objects return `false`. Use `ishghandle` to test for Handle Graphics objects.

Instances of MATLAB fundamental classes return `false`. Use `isa` to test for any of these types. See "Fundamental MATLAB Classes" for more on these classes.

Examples

These examples show the values `isobject` returns when passed specific types:

Define the following MATLAB class:

```
classdef button < handle
   properties
      UiHandle
   end
   methods
      function obj = button(pos)
         obj.UiHandle = uicontrol('Position',pos,...
            'Style','pushbutton');
      end
   end
end
```

Determine which objects are MATLAB objects. For example:

```
h = button([20 20 60 60]);
isobject(h)
ans =
     1
isobject(h.UiHandle)
```

```
ans =
     0
```

Use `isjava` to test for Java objects in MATLAB, where it returns `false` for MATLAB objects:

```
isjava(h)
ans =
     0
```

Create an object that is a MATLAB numeric type:

```
a = pi;
isa(a,'double')
ans =
     1
isobject(a)
ans =
     0
```

See Also

`class` | `isa` | `is*`

Tutorials

- “Class Syntax Fundamentals”

Purpose

Compute isosurface end-cap geometry

Syntax

```
fvc = isocaps(X,Y,Z,V,isovalue)
fvc = isocaps(V,isovalue)
fvc = isocaps(...,'enclose')
fvc = isocaps(...,'whichplane')
[f,v,c] = isocaps(...)
isocaps(...)
```

Description

`fvc = isocaps(X,Y,Z,V,isovalue)` computes isosurface end-cap geometry for the volume data `V` at isosurface value `isovalue`. The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`.

The struct `fvc` contains the face, vertex, and color data for the end-caps and can be passed directly to the `patch` command.

`fvc = isocaps(V,isovalue)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`fvc = isocaps(...,'`*`enclose`*`')` specifies whether the end-caps enclose data values above or below the value specified in `isovalue`. The string *`enclose`* can be either `above` (default) or `below`.

`fvc = isocaps(...,'`*`whichplane`*`')` specifies on which planes to draw the end-caps. Possible values for *`whichplane`* are `all` (default), `xmin`, `xmax`, `ymin`, `ymax`, `zmin`, or `zmax`.

`[f,v,c] = isocaps(...)` returns the face, vertex, and color data for the end-caps in three arrays instead of the struct `fvc`.

`isocaps(...)` without output arguments draws a patch with the computed faces, vertices, and colors.

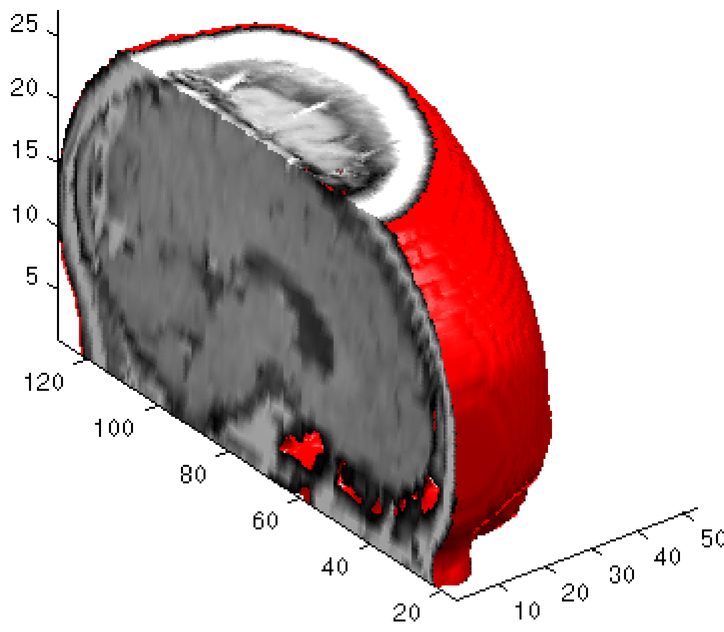
Examples

This example uses a data set that is a collection of MRI slices of a human skull. It illustrates the use of `isocaps` to draw the end-caps on this cutaway volume.

The red `isosurface` shows the outline of the volume (skull) and the end-caps show what is inside of the volume.

The `patch` created from the end-cap data (`p2`) uses interpolated face coloring, which means the gray `colormap` and the light sources determine how it is colored. The isosurface patch (`p1`) used a flat red face color, which is affected by the lights, but does not use the colormap.

```
load mri
D = squeeze(D);
D(:,1:60,:) = [];
p1 = patch(isosurface(D, 5),'FaceColor','red',...
 'EdgeColor','none');
p2 = patch(isocaps(D, 5),'FaceColor','interp',...
 'EdgeColor','none');
view(3); axis tight; daspect([1,1,.4])
colormap(gray(100))
camlight left; camlight; lighting gouraud
isonormals(D,p1)
```



See Also

`isosurface` | `isonormals` | `smooth3` | `subvolume` | `reducevolume` | `reducepatch`

How To

- "Isocaps Add Context to Visualizations"

isocolors

Purpose

Calculate isosurface and patch colors

Syntax

```
nc = isocolors(X,Y,Z,C,vertices)
nc = isocolors(X,Y,Z,R,G,B,vertices)
nc = isocolors(C,vertices)
nc = isocolors(R,G,B,vertices)
nc = isocolors(...,PatchHandle)
isocolors(...,PatchHandle)
```

Description

`nc = isocolors(X,Y,Z,C,vertices)` computes the colors of isosurface (patch object) vertices (`vertices`) using color values `C`. Arrays `X`, `Y`, `Z` define the coordinates for the color data in `C` and must be monotonic vectors that represent a Cartesian, axis-aligned grid (as if produced by `meshgrid`). The colors are returned in `nc`. `C` must be 3-D (index colors).

`nc = isocolors(X,Y,Z,R,G,B,vertices)` uses `R`, `G`, `B` as the red, green, and blue color arrays (true color).

`nc = isocolors(C,vertices)`, and `nc = isocolors(R,G,B,vertices)` assume `X`, `Y`, and `Z` are determined by the expression

```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(C)`.

`nc = isocolors(...,PatchHandle)` uses the vertices from the patch identified by `PatchHandle`.

`isocolors(...,PatchHandle)` sets the `FaceVertexCData` property of the patch specified by `PatchHandle` to the computed colors.

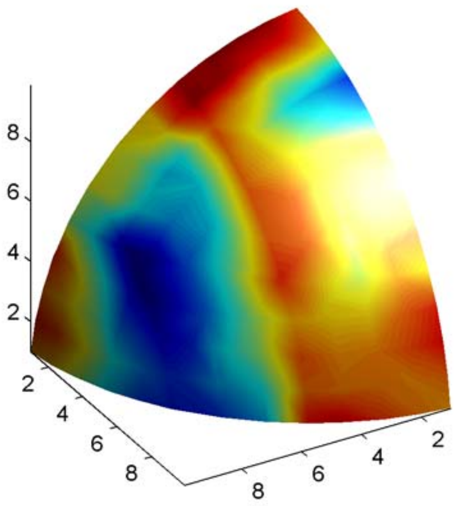
Examples

Indexed Color Data

This example displays an isosurface and colors it with random data using indexed color. (See “Interpolating in Indexed Color Versus Truecolor” for information on how patch objects interpret color data.)

```
[x y z] = meshgrid(1:20,1:20,1:20);
```

```
data = sqrt(x.^2 + y.^2 + z.^2);
cdata = smooth3(rand(size(data)),'box',7);
p = patch(isosurface(x,y,z,data,10));
isonormals(x,y,z,data,p);
isocolors(x,y,z,cdata,p);
set(p,'FaceColor','interp','EdgeColor','none')
view(150,30); daspect([1 1 1]);axis tight
camlight; lighting phong;
```

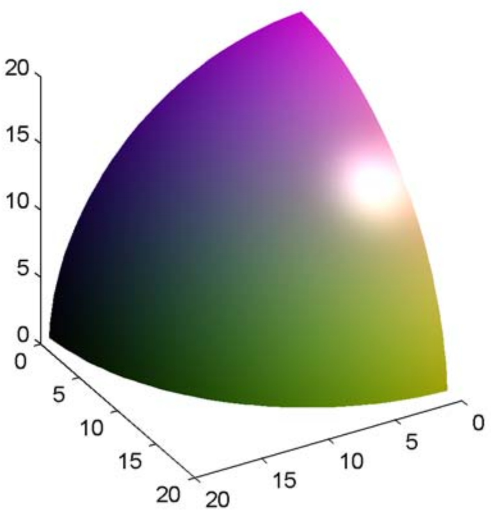


True Color Data

This example displays an isosurface and colors it with true color (RGB) data.

```
[x y z] = meshgrid(1:20,1:20,1:20);
data = sqrt(x.^2 + y.^2 + z.^2);
p = patch(isosurface(x,y,z,data,20));
isonormals(x,y,z,data,p);
[r g b] = meshgrid(20:-1:1,1:20,1:20);
```

```
isocolors(x,y,z,r/20,g/20,b/20,p);
set(p,'FaceColor','interp','EdgeColor','none')
view(150,30); daspect([1 1 1]);
camlight; lighting phong;
```

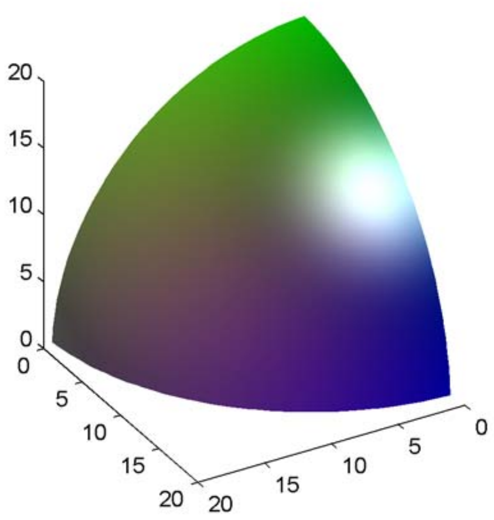


Modified True Color Data

This example uses `isocolors` to calculate the true color data using the isosurface's (patch object's) vertices, but then returns the color data in a variable (`c`) in order to modify the values. It then explicitly sets the isosurface's `FaceVertexCData` to the new data (`1-c`).

```
[x y z] = meshgrid(1:20,1:20,1:20);
data = sqrt(x.^2 + y.^2 + z.^2);
p = patch(isosurface(data,20));
isonormals(data,p);
[r g b] = meshgrid(20:-1:1,1:20,1:20);
c = isocolors(r/20,g/20,b/20,p);
set(p,'FaceVertexCData',1-c)
```

```
set(p,'FaceColor','interp','EdgeColor','none')
view(150,30); daspect([1 1 1]);
camlight; lighting phong;
```



See Also

isosurface | isocaps | smooth3 | subvolume | reducevolume | reducepatch | isonormals

isonormals

Purpose

Compute normals of isosurface vertices

Syntax

```
n = isonormals(X,Y,Z,V,vertices)
n = isonormals(V,vertices)
n = isonormals(V,p) and n = isonormals(X,Y,Z,V,p)
n = isonormals(...,'negate')
isonormals(V,p) and isonormals(X,Y,Z,V,p)
```

Description

`n = isonormals(X,Y,Z,V,vertices)` computes the normals of the isosurface vertices from the vertex list, `vertices`, using the gradient of the data `V`. The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The computed normals are returned in `n`.

`n = isonormals(V,vertices)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`n = isonormals(V,p)` and `n = isonormals(X,Y,Z,V,p)` compute normals from the vertices of the patch identified by the handle `p`.

`n = isonormals(...,'negate')` negates (reverses the direction of) the normals.

`isonormals(V,p)` and `isonormals(X,Y,Z,V,p)` set the `VertexNormals` property of the patch identified by the handle `p` to the computed normals rather than returning the values.

Examples

This example compares the effect of different surface normals on the visual appearance of lit isosurfaces. In one case, the triangles used to draw the isosurface define the normals. In the other, the `isonormals` function uses the volume data to calculate the vertex normals based on the gradient of the data points. The latter approach generally produces a smoother-appearing isosurface.

Define a 3-D array of volume data (`cat`, `interp3`):

```
data = cat(3, [0 .2 0; 0 .3 0; 0 0 0], ...
              [.1 .2 0; 0 1 0; .2 .7 0],...
              [0 .4 .2; .2 .4 0;.1 .1 0]);
```

```
data = interp3(data,3,'cubic');
```

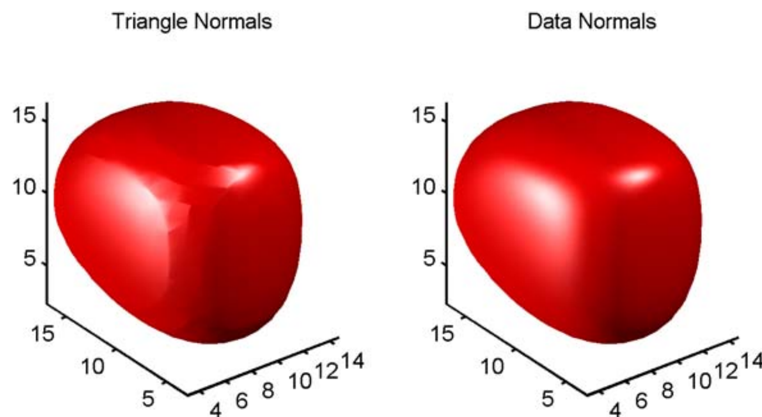
Draw an isosurface from the volume data and add lights. This isosurface uses triangle normals (`patch`, `isosurface`, `view`, `daspect`, `axis`, `camlight`, `lighting`, `title`):

```
subplot(1,2,1)
p1 = patch(isosurface(data,.5),...
'FaceColor','red','EdgeColor','none');
view(3); daspect([1,1,1]); axis tight
camlight; camlight(-80,-10); lighting phong;
title('Triangle Normals')
```

Draw the same lit isosurface using normals calculated from the volume data:

```
subplot(1,2,2)
p2 = patch(isosurface(data,.5),...
    'FaceColor','red','EdgeColor','none');
isonormals(data,p2)
view(3); daspect([1 1 1]); axis tight
camlight;  camlight(-80,-10); lighting phong;
title('Data Normals')
```

These isosurfaces illustrate the difference between triangle and data normals:



See Also

interp3 | isosurface | isocaps | smooth3 | subvolume | reducevolume | reducepatch

Purpose

Extract isosurface data from volume data

Syntax

```
fv = isosurface(X,Y,Z,V,isovalue)
fv = isosurface(V,isovalue)
fvc = isosurface(...,colors)
fv = isosurface(...,'noshare')
fv = isosurface(...,'verbose')
[f,v] = isosurface(...)
[f,v,c] = isosurface(...)
isosurface(...)
```

Description

`fv = isosurface(X,Y,Z,V,isovalue)` computes isosurface data from the volume data `V` at the isosurface value specified in `isovalue`. That is, the isosurface connects points that have the specified value much the way contour lines connect points of equal elevation.

The arrays `X`, `Y`, and `Z` represent a Cartesian, axis-aligned grid. `V` contains the corresponding values at these grid points. The coordinate arrays (`X`, `Y`, and `Z`) must be monotonic and conform to the format produced by `meshgrid`. `V` must be a 3D volume array of the same size as `X`, `Y`, and `Z`.

The `struct` `fv` contains the faces and vertices of the isosurface, which you can pass directly to the `patch` command.

`fv = isosurface(V,isovalue)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)` where `[m,n,p] = size(V)`.

`fvc = isosurface(...,colors)` interpolates the array `colors` onto the scalar field and returns the interpolated values in the `facevertexcdata` field of the `fvc` structure. The size of the `colors` array must be the same as `V`. The `colors` argument enables you to control the color mapping of the isosurface with data different from that used to calculate the isosurface (e.g., temperature data superimposed on a wind current isosurface).

fv = isosurface(...,'noshare') does not create shared vertices. This is faster, but produces a larger set of vertices.

fv = isosurface(...,'verbose') prints progress messages to the command window as the computation progresses.

[f,v] = isosurface(...) or [f,v,c] = isosurface(...) returns the faces and vertices (and faceVertexcCData) in separate arrays instead of a struct.

isosurface(...) with no output arguments, creates a patch in the current axes with the computed faces and vertices. If no current axes exists, a new axes is created with a 3-D view and appropriate lighting.

Special Case Behavior — isosurface Called with No Output Arguments

If there is no current axes and you call isosurface with without assigning output arguments, MATLAB creates a new axes, sets it to a 3-D view, and adds lighting to the isosurface graph.

Tips

You can pass the fv structure created by isosurface directly to the patch command, but you cannot pass the individual faces and vertices arrays (f, v) to patch without specifying property names. For example,

```
patch(isosurface(X,Y,Z,V,isovalue))
```

or

```
[f,v] = isosurface(X,Y,Z,V,isovalue);
patch('Faces',f,'Vertices',v)
```

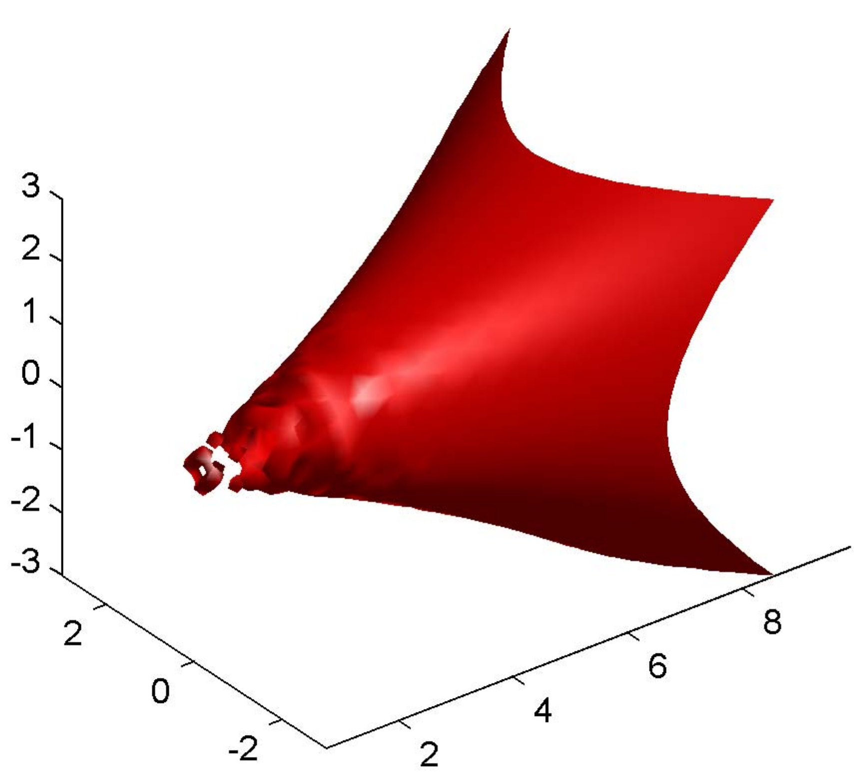
Examples

Example 1

This example uses the flow data set, which represents the speed profile of a submerged jet within an infinite tank (type help flow for more information). The isosurface is drawn at the data value of -3. The statements that follow the patch command prepare the isosurface for lighting by

- Recalculating the isosurface normals based on the volume data (`isonormals`)
- Setting the face and edge color (`set`, `FaceColor`, `EdgeColor`)
- Specifying the view (`daspect`, `view`)
- Adding lights (`camlight`, `lighting`)

```
[x,y,z,v] = flow;
p = patch(isosurface(x,y,z,v,-3));
isonormals(x,y,z,v,p)
set(p,'FaceColor','red','EdgeColor','none');
daspect([1,1,1])
view(3); axis tight
camlight
lighting gouraud
```

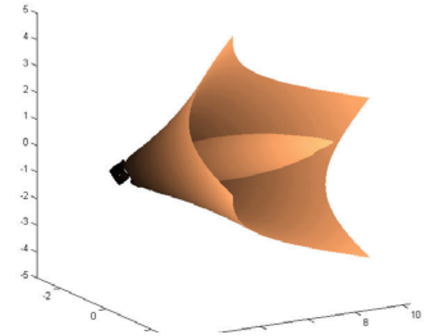


Example 2

Visualize the same flow data as above, but color-code the surface to indicate magnitude along the X-axis. Use a sixth argument to `isosurface`, which provides a means to overlay another data set by coloring the resulting isosurface. The `colors` variable is a vector containing a scalar value for each vertex in the isosurface, to be portrayed with the current color map. In this case, it is one of the

variables that define the surface, but it could be entirely independent. You can apply a different color scheme by changing the current figure color map.

```
[x,y,z,v] = flow;
[faces,verts,colors] = isosurface(x,y,z,v,-3,x);
patch('Vertices', verts, 'Faces', faces, ...
    'FaceVertexCData', colors, ...
    'FaceColor','interp', ...
    'edgecolor', 'interp');
view(30,-15);
axis vis3d;
colormap copper
```



See Also

isonormals | shrinkfaces | smooth3 | subvolume

How To

- “Connecting Equal Values with Isosurfaces”

ispc

Purpose

Determine if version is for Windows (PC) platform

Syntax

```
tf = ispc
```

Description

`tf = ispc` returns logical `1` (`true`) if the version of MATLAB software is for the Microsoft Windows platform, and returns logical `0` (`false`) otherwise.

See Also

`isunix` | `ismac` | `computer` | `isstudent` | `is*`

Purpose

Test for existence of preference

Syntax

```
ispref('group','pref')
ispref('group')
ispref('group',{'pref1','pref2',...'prefn'})
```

Description

`ispref('group','pref')` returns `1` if the preference specified by `group` and `pref` exists, and `0` otherwise.

`ispref('group')` returns `1` if the GROUP exists, and `0` otherwise.

`ispref('group',{'pref1','pref2',...'prefn'})` returns a logical array the same length as the cell array of preference names, containing `1` where each preference exists, and `0` elsewhere.

Examples

```
addpref('mytoolbox','version','1.0')
ispref('mytoolbox','version')

ans =
     1.0
```

See Also

`addpref` | `getpref` | `rmpref` | `setpref` | `uigetpref` | `uisetpref`

isprime

Purpose

Determine which array elements are prime

Syntax

```
TF = isprime(X)
```

Description

`TF = isprime(X)` returns a logical array the same size as `X`. The value at `TF(i)` is `true` when `X(i)` is a prime number. Otherwise, the value is `false`.

Input Arguments

X - Input values

scalar, vector, or array of real, nonnegative integer values

Input values, specified as a scalar, vector, or array of real, nonnegative integer values.

Example: `17`

Example: `[1 2 3 4]`

Example: `int16([127 255 4095])`

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Examples

Determine if Double Integer Values Are Prime

```
tf = isprime([2 3 0 6 10])

tf =

     1     1     0     0     0
```

`2` and `3` are prime, but `0`, `6`, and `10` are not.

Determine if Unsigned Integer Values Are Prime

```
x = uint16([333 71 99]);
tf = isprime(x)

tf =
```

```
     0     1     0
```

`71` is prime, but `333` and `99` are not.

See Also

`is*` | `primes`

isprop

Purpose

Determine if property of object

Syntax

```
tf = isprop(obj,'PropertyName')
```

Description

`tf = isprop(obj,'PropertyName')` returns logical `1` (`true`) if the specified `PropertyName` is a property of object `obj`. Otherwise, returns logical `0` (`false`).

Note that classes can control access to property values by defining property attributes. Property access can be defined as:

- Readable and writable
- Read only
- Write only
- Accessible only to certain class methods

See "Property Attributes" for information on restrictions to property access.

See "Getting Information About Properties" for information about obtaining specific information about the properties of a class.

Examples

This example uses `isprop` to determine if `XDataSource` is a property of object `h` before attempting to set the property value:

```
x = 0:pi/10:2*pi;
y = sin(x);
h = plot(x,y);
if isprop(h,'XDataSource')
   set(h,'XDataSource','x')
else
   error(['XDataSource is not a member of class ',class(h)])
end
```

See Also

`properties` | `ismethod`

Tutorials

- “Properties”

isprop (COM)

Purpose

Determine whether input is COM object property

Syntax

```
tf = h.isprop('propertyname')
tf = isprop(h, 'propertyname')
```

Description

`tf = h.isprop('propertyname')` returns logical `1` (`true`) if the specified `name` is a property of COM object `h`. Otherwise, returns logical `0` (`false`).

`tf = isprop(h, 'propertyname')` is an alternate syntax.

COM functions are available on Microsoft Windows systems only.

Examples

Test a property of an instance of a Microsoft Excel application:

```
h = actxserver('Excel.Application');
isprop(h,'UsableWidth')
```

MATLAB returns `true`, `UsableWidth` is a property.

Try the same test on `SaveWorkspace`:

```
isprop(h,'SaveWorkspace')
```

MATLAB returns `false`. `SaveWorkspace` is not a property; it is a method.

See Also

`inspect` | `ismethod` | `isevent`

How To

- "Exploring Properties"

Purpose

Check if input is real array

Syntax

```
TF = isreal(A)
```

Description

`TF = isreal(A)` returns logical `1` (`true`) if `A` does not have an imaginary part. It returns logical `0` (`false`) otherwise. If `A` has a stored imaginary part of value `0`, `isreal(A)` returns logical `0` (`false`).

Note For `logical` and `char` data classes, `isreal` always returns `true`. For numeric data types, if `A` does not have an imaginary part `isreal` returns `true`; if `A` does have an imaginary part `isreal` returns `false`. For `cell`, `struct`, `function_handle`, and `object` data types, `isreal` always returns `false`.

`~isreal(x)` returns `true` for arrays that have at least one element with an imaginary component. The value of that component can be `0`.

Tips

If `A` is real, `complex(A)` returns a complex number whose imaginary component is `0`, and `isreal(complex(A))` returns `false`. In contrast, the addition `A + 0i` returns the real value `A`, and `isreal(A + 0i)` returns `true`.

If `B` is real and `A = complex(B)`, then `A` is a complex matrix and `isreal(A)` returns `false`, while `A(m:n)` returns a real matrix and `isreal(A(m:n))` returns `true`.

Because MATLAB software supports complex arithmetic, certain of its functions can introduce significant imaginary components during the course of calculations that appear to be limited to real numbers. Thus, you should use `isreal` with discretion.

Examples

Example 1

If a computation results in a zero-value imaginary component, `isreal` returns `true`.

```
x=3+4i;
y=5-4i;
isreal(x+y)

ans =

     1
```

Example 2

These examples use `isreal` to detect the presence or absence of imaginary numbers in an array. Let

```
x = magic(3);
y = complex(x);
```

`isreal(x)` returns `true` because no element of `x` has an imaginary component.

```
isreal(x)

ans =

     1
```

`isreal(y)` returns `false`, because every element of `x` has an imaginary component, even though the value of the imaginary components is `0`.

```
isreal(y)

ans =

     0
```

This expression detects strictly real arrays, i.e., elements with `0`-valued imaginary components are treated as real.

```
~any(imag(y(:)))

ans =
```

```
     1
```

Example 3

Given the following cell array,

```
C{1} = pi;                  % double
C{2} = 'John Doe';          % char array
C{3} = 2 + 4i;              % complex double
C{4} = ispc;                % logical
C{5} = magic(3);            % double array
C{6} = complex(5,0)         % complex double

C =

    [3.1416]    'John Doe'    [2.0000 + 4.0000i]    [1]    [3x3 double
```

isreal shows that all but C{1,3} and C{1,6} are real arrays.

```
for k = 1:6
x(k) = isreal(C{k});
end

x

x =

     1     1     0     1     1     0
```

See Also

complex | isnumeric | isnan | isprime | isfinite | isinf | isa | is*

Purpose

Determine whether input is row vector

Syntax

`isrow(V)`

Description

`isrow(V)` returns logical 1 (`true`) if `size(V)` returns `[1 n]` with a nonnegative integer value `n`, and logical 0 (`false`) otherwise.

Examples

Determine if a vector is a row. This example is a column so `isrow` returns 0:

```
V = rand(5,1);
isrow(V)
ans =
     0
```

Transpose the vector to make it a row. `isrow` returns 1:

```
V1 = V';
isrow(V1)
ans =
     1
```

See Also

`iscolumn` | `ismatrix` | `isscalar` | `isvector`

Purpose Determine whether input is scalar

Syntax

```
isscalar(A)
```

Description `isscalar(A)` returns logical `1` (`true`) if `size(A)` returns [1 1], and logical `0` (`false`) otherwise.

Examples Test matrix `A` and one element of the matrix:

```
A = rand(5);

isscalar(A)
ans =
     0

isscalar(A(3,2))
ans =
     1
```

See Also `isvector` | `ismatrix` | `isrow` | `iscolumn` | `islogical` | `ischar` | `isa` | `is*`

issorted

Purpose

Determine whether set elements are in sorted order

Syntax

```
TF = issorted(A)
TF = issorted(A, 'rows')
```

Description

`TF = issorted(A)` returns logical `1` (`true`) if the elements of `A` are in sorted order, and logical `0` (`false`) otherwise. Input `A` can be a vector or an N-by-1 or 1-by-N cell array of strings. `A` is considered to be sorted if `A` and the output of `sort(A)` are equal.

`TF = issorted(A, 'rows')` returns logical `1` (`true`) if the rows of two-dimensional matrix `A` are in sorted order, and logical `0` (`false`) otherwise. Matrix `A` is considered to be sorted if `A` and the output of `sortrows(A)` are equal.

Note Only the `issorted(A)` syntax supports `A` as a cell array of strings.

Tips

For character arrays, `issorted` uses ASCII, rather than alphabetical, order.

You cannot use `issorted` on arrays of greater than two dimensions.

Examples

Example 1 — Using issorted on a vector

```
A = [5 12 33 39 78 90 95 107 128 131];

issorted(A)
ans =
     1
```

Example 2 — Using issorted on a matrix

```
A = magic(5)
A =
    17    24     1     8    15
    23     5     7    14    16
```

```
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

issorted(A, 'rows')
ans =
     0

B = sortrows(A)
B =
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
    17    24     1     8    15
    23     5     7    14    16

issorted(B)
ans =
     1
```

Example 3 — Using issorted on a cell array

```
x = {'one'; 'two'; 'three'; 'four'; 'five'};
issorted(x)
ans =
     0

y = sort(x)
y =
    'five'
    'four'
    'one'
    'three'
    'two'

issorted(y)
```

See Also `sort` | `sortrows` | `ismember` | `unique` | `intersect` | `union` | `setdiff` | `setxor` | `is*`

Purpose

Array elements that are space characters

Syntax

```
tf = isspace('str')
```

Description

`tf = isspace('str')` returns an array the same size as `'str'` containing logical `1` (`true`) where the elements of `str` are Unicode-represented whitespace characters, and logical `0` (`false`) where they are not.

Whitespace characters for which `isspace` returns `true` include tab, line feed, vertical tab, form feed, carriage return, and space, in addition to a number of other Unicode characters. To see all characters for which `isspace` returns `true`, enter the following command, and then look up the returned decimal codes in a Unicode reference:

```
find(isspace(char(1):char(intmax('uint16'))))
```

Examples

```
isspace('  Find spa ces ')
  Columns 1 through 13
     1   1   0   0   0   0   1   0   0   0   1   0   0
  Columns 14 through 15
     0   1
```

See Also

`isletter` | `isstrprop` | `ischar` | `strings` | `isa` | `is*`

issparse

Purpose

Determine whether input is sparse

Syntax

```
TF = issparse(S)
```

Description

`TF = issparse(S)` returns logical 1 (`true`) if the storage class of `S` is sparse and logical 0 (`false`) otherwise.

See Also

`is*` | `sparse` | `full`

Purpose

Determine whether input is character array

Note `isstr` is not recommended. Use `ischar` instead.

See Also

`ischar` | `isa` | `is*`

isstrprop

Purpose

Determine whether string is of specified category

Syntax

```
tf = isstrprop('str', 'category')
```

Description

`tf = isstrprop('str', 'category')` returns a logical array the same size as `str` containing logical `1` (`true`) where the elements of `str` belong to the specified *`category`*, and logical `0` (`false`) where they do not.

The `str` input can be a character array, cell array, or any MATLAB numeric type. If `str` is a cell array, then the return value is a cell array of the same shape as `str`.

The *`category`* input can be any of the strings shown in the left column below:

Category	Description
`alpha`	`True` for those elements of `str` that are alphabetic
`alphanum`	`True` for those elements of `str` that are alphanumeric
`cntrl`	`True` for those elements of `str` that are control characters (for example, `char(0:20)`)
`digit`	`True` for those elements of `str` that are numeric digits

`graphic`	True for those elements of str that are graphic characters. These are all values that represent any characters except for the following:
	
	`unassigned, space, line separator,`
	`paragraph separator, control characters,`
	`Unicode format control characters,`
	`private user-defined characters,`
	`Unicode surrogate characters,`
	`Unicode other characters`

`lower`	`True` for those elements of `str` that are lowercase letters
`print`	`True` for those elements of `str` that are graphic characters, plus `char(32)`

Category	Description
`punct`	`True` for those elements of `str` that are punctuation characters
`wspace`	`True` for those elements of `str` that are white-space characters. This range includes the ANSI® C definition of white space, `{' ','\t','\n','\r','\v','\f'}`, in addition to a number of other Unicode characters.
`upper`	`True` for those elements of `str` that are uppercase letters
`xdigit`	`True` for those elements of `str` that are valid hexadecimal digits

Tips

Numbers of type `double` are converted to `int32` according to MATLAB rules of double-to-integer conversion. Numbers of type `int64` and `uint64` bigger than `int32(inf)` saturate to `int32(inf)`.

MATLAB classifies the elements of the `str` input according to the Unicode definition of the specified category. If the numeric value of an element in the input array falls within the range that defines a Unicode character category, then this element is classified as being of that category. The set of Unicode character codes includes the set of ASCII character codes, but also covers a large number of languages beyond the scope of the ASCII set. The classification of characters is dependent on the global location of the platform on which MATLAB is installed.

Whitespace characters for which the `wspace` option returns `true` include tab, line feed, vertical tab, form feed, carriage return, and space, in addition to a number of other Unicode characters. To see all characters for which the `wspace` option returns `true`, enter the following command, and then look up the returned decimal codes in a Unicode reference:

```
find(isstrprop(char(1):char(intmax('uint16')), 'wspace'))
```

Examples

Test for alphabetic characters in a string:

```
A = isstrprop('abc123def', 'alpha')
```

```
A =
   1  1  1  0  0  0  1  1  1
```

Test for numeric digits in a string:

```
A = isstrprop('abc123def', 'digit')
A =
   0  0  0  1  1  1  0  0  0
```

Test for hexadecimal digits in a string:

```
A = isstrprop('abcd1234efgh', 'xdigit')
A =
   1  1  1  1  1  1  1  1  1  1  0  0
```

Test for numeric digits in a character array:

```
A = isstrprop(char([97 98 99 49 50 51 101 102 103]), ...
              'digit')
A =
   0  0  0  1  1  1  0  0  0
```

Test for alphabetic characters in a two-dimensional cell array:

```
A = isstrprop({'abc123def';'456ghi789'}, 'alpha')
A =
   [1x9 logical]
   [1x9 logical]

A{:,:}
ans =
   1  1  1  0  0  0  1  1  1
   0  0  0  1  1  1  0  0  0
```

Test for white-space characters in a string:

```
A = isstrprop(sprintf('a bc\n'), 'wspace')
A =
   0  1  0  0  1
```

See Also

`strings` | `ischar` | `isletter` | `isspace` | `iscellstr` | `isnumeric` | `isa` | `is*`

isstruct

Purpose

Determine whether input is structure array

Syntax

```
tf = isstruct(A)
```

Description

`tf = isstruct(A)` returns logical `1` (`true`) if `A` is a MATLAB structure and logical `0` (`false`) otherwise.

Examples

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];

isstruct(patient)

ans =

     1
```

See Also

`struct` | `isfield` | `iscell` | `ischar` | `isobject` | `isnumeric` | `islogical` | `isa` | `is*`

How To

- dynamic field names

Purpose Determine if version is Student Version

Syntax

```
tf = isstudent
```

Description `tf = isstudent` returns logical `1` (`true`) if the version of MATLAB software is the Student Version, and returns logical `0` (`false`) for commercial versions.

See Also `ver` | `version` | `license` | `ispc` | `isunix` | `is*`

Tiff.isTiled

Purpose

Determine if tiled image

Syntax

```
bool = tiffobj.isTiled()
```

Description

`bool = tiffobj.isTiled()` returns `true` if the image has a tiled organization and `false` if the image has a stripped organization.

Examples

Open a `Tiff` object and check if the image in the TIFF file has a tiled or stipped organization. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path.

```
t = Tiff('myfile.tif', 'r');
tf = t.isTiled();
```

References

This method corresponds to the `TIFFIsTiled` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

Purpose

Determine if version is for UNIX platform

Syntax

```
tf = isunix
```

Description

`tf = isunix` returns logical `1` (`true`) if the version of MATLAB software is for the UNIX platform or the Apple Mac OS X platform, and returns logical `0` (`false`) otherwise.

See Also

`ispc` | `ismac` | `computer` | `isstudent` | `is*`

isvalid (handle)

Purpose

Is object valid handle class object

Syntax

```
Hl = isvalid(Hobj)
```

Description

`Hl = isvalid(Hobj)` returns a logical array (or scalar if `Hobj` is scalar) in which each element is `true` if the corresponding element in `Hobj` is a valid handle. This method is `Sealed`, so you cannot override it in a handle subclass.

Note This method does not work with Handle Graphics objects. To determine the validity of a Handle Graphics object handle, use the `ishghandle` function.

See Also

`delete (handle)` | `handle`

How To

- "Testing Handle Validity"

Purpose

Determine whether serial port objects are valid

Syntax

```
out = isvalid(obj)
```

Description

`out = isvalid(obj)` returns the logical array `out`, which contains a `0` where the elements of the serial port object, `obj` are invalid serial port objects and a `1` where the elements of `obj` are valid serial port objects.

Tips

`obj` becomes invalid after it is removed from memory with the `delete` function. Because you cannot connect an invalid serial port object to the device, you should remove it from the workspace with the `clear` command.

Examples

Suppose you create the following two serial port objects.

```
s1 = serial('COM1');
s2 = serial('COM1');
```

`s2` becomes invalid after it is deleted.

```
delete(s2)
```

`isvalid` verifies that `s1` is valid and `s2` is invalid.

```
sarray = [s1 s2];
isvalid(sarray)
ans =
     1     0
```

See Also

`clear` | `delete`

isvalid (timer)

Purpose Determine whether timer object is valid

Syntax

```
out = isvalid(obj)
```

Description

`out = isvalid(obj)` returns a logical array, `out`, that contains a `0` where the elements of `obj` are invalid timer objects and a `1` where the elements of `obj` are valid timer objects.

An invalid timer object is an object that has been deleted and cannot be reused. Use the `clear` command to remove an invalid timer object from the workspace.

Examples

Create a valid timer object.

```
t = timer;
out = isvalid(t)
out =

    1
```

Delete the timer object, making it invalid.

```
delete(t)
out1 = isvalid(t)
out1 =

    0
```

See Also

`timer` | `delete(timer)`

Purpose

Determine whether input is valid variable name

Syntax

```
tf = isvarname('str')
isvarname str
```

Description

`tf = isvarname('str')` returns logical `1` (`true`) if the string `str` is a valid MATLAB variable name and logical `0` (`false`) otherwise. A valid variable name is a character string of letters, digits, and underscores, totaling not more than `namelengthmax` characters and beginning with a letter.

MATLAB keywords are not valid variable names. Type the command `iskeyword` with no input arguments to see a list of MATLAB keywords.

`isvarname str` uses the MATLAB command format.

Examples

This variable name is valid:

```
isvarname foo
ans =
   1
```

This one is not because it starts with a number:

```
isvarname 8th_column
ans =
   0
```

If you are building strings from various pieces, place the construction in parentheses.

```
d = date;

isvarname(['Monday_', d(1:2)])
ans =
   1
```

See Also

`genvarname` | `isglobal` | `iskeyword` | `namelengthmax` | `is*`

isvector

Purpose

Determine whether input is vector

Syntax

```
isvector(A)
```

Description

`isvector(A)` returns logical `1` (`true`) if size(A) returns `[1 n]` or `[n 1]` with a nonnegative integer value `n`, and logical `0` (`false`) otherwise.

Examples

Test matrix `A` and its row and column vectors:

```
A = rand(5);

isvector(A)
ans =
     0

isvector(A(3, :))
ans =
     1

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

See Also

`iscolumn` | `ismatrix` | `isrow` | `isscalar` | `isempty` | `isnumeric` | `islogical` | `ischar` | `isa` | `is*`

Purpose

Imaginary unit

Syntax

```
j
x+yj
x+j*y
```

Description

As the basic imaginary unit `sqrt(-1)`, `j` is used to enter complex numbers. You can also use the character `i` as the imaginary unit. If you want to create a complex number without using `i` and `j`, you can use the `complex` function.

Since `j` is a function, it can be overridden and used as a variable. However, it is best to avoid using `i` and `j` for variable names if you intend to use them in complex arithmetic.

If desired, you can use the character `j` without a multiplication sign as a suffix in forming a complex numerical constant.

For speed and improved robustness, you can replace complex `i` and `j` by `1i`. For example, instead of using

```
a = j;
```

use

```
a = 1i;
```

Examples

```
Z = 2+3j
Z = x+j*y
Z = r*exp(j*theta)
```

See Also

`conj` | `i` | `imag` | `real`

javaaddpath

Purpose

Add entries to dynamic Java class path

Syntax

```
javaaddpath(dpath)
javaaddpath(dpath,'-end')
```

Description

`javaaddpath(dpath)` adds one or more folders or Java Archive (JAR) files to the beginning of the current dynamic class path.

`javaaddpath(dpath,'-end')` adds files or folders to the end of the path.

Tips

- MATLAB calls the `clear java` command whenever you change the dynamic path. This command clears the definitions of all Java classes defined by files on the dynamic class path, removes all variables from the base workspace, and removes all compiled scripts, functions, and MEX-functions from memory.

Input Arguments

dpath - Folder or JAR file

string | cell array of strings

Folder or JAR file, specified as a string or cell array of strings, to add to the dynamic path. When you add a folder to the path, MATLAB includes all files in that folder as part of the path.

Data Types

`char` | `cell`

Examples

Add Folder to Dynamic Class Path

Display the current dynamic path.

```
javaclasspath('-dynamic')
```

```
  DYNAMIC JAVA PATH

    <empty>
```

The output reflects your configuration.

Add the current folder.

```
javaaddpath(pwd)
```

Display the dynamic path.

```
p = javaclasspath

p =
    'c:\work\Java'
```

The output reflects your current folder.

Append URL to Dynamic Class Path

```
javaaddpath('http://www.example.com','-end')
p = javaclasspath

p =
    'c:\work\Java'
    'http://www.example.com'
```

See Also

javaclasspath | javarmpath | clear

Concepts

- "Bringing Java Classes into MATLAB Workspace"

javaArray

Purpose

Construct Java array object

Syntax

```
ObjArr = javaArray(PackageName.ClassName,x1,...,xN)
```

Description

`ObjArr = javaArray(PackageName.ClassName,x1,...,xN)` constructs an empty Java array object for objects of the specified `PackageName.ClassName` class.

Input Arguments

PackageName.ClassName - Name of Java class
string

Name of Java class, including package name, specified as a string.

Data Types
`char`

x1,...,xN - Dimensions of the array
integer

Dimensions of the array, specified as integer.

Data Types
`double`

Output Arguments

ObjArr - Java array

Java array with dimensions `x1,...,xN`.

Definitions

Java Array Object

A Java array object is an object with Java dimensionality.

Tips

- The array created by `javaArray` is equivalent to the array created by the following Java code:

  ```
  A = new PackageName.ClassName[x1]...[xN];
  ```

- To create an array of primitive Java types, create an array of the equivalent MATLAB type, as shown in the table, "Conversion of MATLAB Types to Java Types".

Examples

Create 4-By-5 Array

Create 4-by-5 array of `java.lang.Double` type.

```
x1 = 4; x2 = 5;
dblArray = javaArray ('java.lang.Double',x1,x2);
```

Fill in values.

```
for m = 1:x1
    for n = 1:x2
        dblArray(m,n) = java.lang.Double((m*10) + n);
    end
end
```

Display results.

```
dblArray
```

```
dblArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]    [15]
    [21]    [22]    [23]    [24]    [25]
    [31]    [32]    [33]    [34]    [35]
    [41]    [42]    [43]    [44]    [45]
```

See Also

`javaObjectEDT` | `javaMethodEDT` | `class` | `methodsview` | `isjava`

Concepts

- "Working with Java Arrays"

javachk

Purpose

Error message based on Java feature support

Syntax

```
MSG = javachk(feature)
javachk(feature,component)
```

Description

`MSG = javachk(feature)` returns a generic error message if the specified Java feature is not available in the current MATLAB session.

`javachk(feature,component)` also names the specified `component` in the error message.

Input Arguments

feature - Java feature

`'awt'` | `'desktop'` | `'jvm'` | `'swing'`

Java feature, specified as one of these values:

`'awt'`	Java GUI components in the Abstract Window Toolkit (AWT) components are available.
`'desktop'`	The MATLAB interactive desktop is running.
`'jvm'`	The Java Virtual Machine software (JVM™) is running.
`'swing'`	Swing components (Java lightweight GUI components in the Java Foundation Classes) are available.

component - Identifier

string

Identifier, specified as a string, to display in the error message.

Data Types

`char`

Output Arguments

MSG - Error message

structure

Error message, returned as a structure with these fields:

If it is available, `javachk` returns an error structure with empty `message` and `identifier` fields.

message - Message

string | empty

Message, specified as a string.

identifier - Identifier

string | empty

Identifier, specified as a string.

Examples

Generate Error

```
if isempty(javachk('jvm'))
   scalar = java.lang.Double(5);
end
% Check that JVM is available & JavaFigures are supported
error(javachk('jvm'))
error(javachk('awt'))
```

Generate Error in User-Defined Script

If you write a script, `myFile`, that displays a Java Frame and want it to error gracefully if a frame cannot be displayed, do the following:

```
error(javachk('awt','myFile'));
myFrame = java.awt.Frame;
myFrame.setVisible(1);
```

If the script cannot display a frame, it displays this error:

```
myFile is not supported on this platform.
```

See Also

usejava | error

Purpose

Return Java class path or specify dynamic path

Syntax

```
javaclasspath
javaclasspath('-dynamic')
javaclasspath('-static')

dpath = javaclasspath
spath = javaclasspath('-static')
jpath = javaclasspath('-all')

javaclasspath(dpath)
javaclasspath(dpath1,dpath2)

javaclasspath(statusmsg)
```

Description

`javaclasspath` displays the static and dynamic segments of the Java class path.

`javaclasspath('-dynamic')` displays the dynamic path.

`javaclasspath('-static')` displays the static path.

`dpath = javaclasspath` returns the dynamic path, `dpath`.

`spath = javaclasspath('-static')` returns the static path, `spath`.

`jpath = javaclasspath('-all')` returns the entire path, `jpath`. The returned cell array contains first the static segment of the path, and then the dynamic segment.

`javaclasspath(dpath)` changes the dynamic path to `dpath`. Use this syntax to reload Java classes.

`javaclasspath(dpath1,dpath2)` changes the dynamic path to the concatenation of paths `dpath1,dpath2`.

`javaclasspath(statusmsg)` enables or disables the display of status messages.

Input Arguments

dpath - Path entries

string | cell array of strings

Path entries, specified as a string or cell array of strings, to specify for the dynamic path. Converts relative paths to absolute paths.

Example: javaclasspath('http://domain.com')

Data Types
`char | cell`

dpath1,dpath2 - Path entries

string | cell array of strings

Path entries, specified as a string or cell array of strings, concatenated, to specify for the dynamic path.

Data Types
`char | cell`

statusmsg - Message flag

'-v0' (default) | '-v1'

Message flag, specified as one of these values:

`'-v0'`	Does not display status messages while loading the Java path from the file system.
`'-v1'`	Displays status messages.

Controls status message display from the `javaclasspath`, `javaaddpath`, and `javarmpath` functions.

Output Arguments

dpath - Dynamic path entries
cell array of strings

Dynamic path entries for the current path, returned as a cell array of strings. If no path entries are defined, `dpath` is an empty cell array.

spath - Static path entries
cell array of strings

Static path entries for the current path, returned as a cell array of strings. If no path entries are defined, `spath` is an empty cell array.

jpath - All path entries
cell array of strings

All path entries, returned as a cell array of strings. If no path entries are defined, `jpath` is an empty cell array.

Tips

- MATLAB searches the static path before the dynamic path.
- Java classes on the static path should not have dependencies on classes on the dynamic path. Such dependencies produce run-time errors.
- MATLAB calls the `clear java` command whenever you change the dynamic path. This command clears the definitions of all Java classes defined by files on the dynamic class path, removes all variables from the base workspace, and removes all compiled scripts, functions, and MEX-functions from memory.
- MATLAB displays a warning if you add an entry to the dynamic path that is already specified on the static path.

Definitions

Static Path

The static path is a segment of the Java path which is loaded at the start of each MATLAB session from the MATLAB built-in Java path and the file `javaclasspath.txt`.

The static Java path offers better Java class-loading performance than the dynamic Java path. However, to modify the static Java path you need to edit the file `javaclasspath.txt` and restart MATLAB.

Dynamic Path

The dynamic path is a segment of the Java path which is loaded any time during a MATLAB session using the `javaclasspath` function.

You can define the dynamic path (using `javaclasspath`), modify the path (using `javaaddpath` and `javarmpath`), and refresh the Java class definitions for all classes on the dynamic path (using `clear java`) without restarting MATLAB.

Examples

Modify Path Using Cell Array

Create a cell array with two path values.

```
dpath = {'http://domain.com','http://some.domain.com/jarfile.jar'};
```

Set message flag to display class-loading messages.

```
javaclasspath('-v1')
```

Modify path.

```
javaclasspath(dpath)
```

```
Loading following class path(s) from local file system:
* http://domain.com
* http://some.domain.com/jarfile.jar
```

Display updated dynamic path.

```
javaclasspath('-dynamic')
```

```
  DYNAMIC JAVA PATH

 http://domain.com
 http://some.domain.com/jarfile.jar
```

MATLAB adds folders from `dpath` to the existing path.

Capture Contents of Dynamic Path

Create a cell array, `p`, with the entries of the dynamic path.

```
javaclasspath('-v0') %Suppress display of class-loading messages
p = javaclasspath

p =

     {}
```

If there are no entries on the dynamic path, MATLAB creates an empty cell array.

See Also

`javaaddpath` | `javarmpath` | `clear`

Concepts

- "The Java Class Path"

matlab.exception.JavaException

Purpose

Capture error information for Java exception

Description

Process information from a `matlab.exception.JavaException` object to handle Java errors thrown from Java methods called from MATLAB. This class is derived from `MException`.

Construction

`e = matlab.exception.JavaException(msgID,errMsg,excObj)` constructs instance `e` of `matlab.exception.JavaException` class.

Input Arguments

msgID

message identifier

errMsg

error message string

excObj

`java.lang.Throwable` object that caused the exception

Output Arguments

e

Instance of `matlab.exception.JavaException` class

Properties

ExceptionObject

Java exception object that caused the error.

Tips

- You do not typically construct a `matlab.exception.JavaException` object explicitly. MATLAB automatically constructs a `JavaException` object whenever Java throws an exception. The `JavaException` object wraps the original Java exception.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

```
try
    java.lang.Class.forName('foo');
catch e
    e.message
    if(isa(e, 'matlab.exception.JavaException'))
        ex = e.ExceptionObject;
        assert(isJava(ex));
        ex.printStackTrace;
    end
end
```

Concepts

- “Capture Information About Exceptions”
- “Throw an Exception”

javaMethod

Purpose

Call Java method

Syntax

```
javaMethod(MethodName,JavaObj,x1,...,xN)
javaMethod(StaticMethodName,ClassName,x1,...,xN)
```

Description

`javaMethod(MethodName,JavaObj,x1,...,xN)` calls the method in the class of the Java object array with the signature matching the arguments `x1,...,xN`.

`javaMethod(StaticMethodName,ClassName,x1,...,xN)` calls the static method in class `ClassName`.

Input Arguments

MethodName - Name of nonstatic Java method

string

Name of nonstatic Java method, specified as a string.

Data Types

`char`

JavaObj - Array

Java object

Array, specified as a Java object of the class containing the method.

x1,...,xN - Java method input arguments

any type

Java method input arguments, 1 through N (if any), required by `MethodName` or `StaticMethodName`, specified by any type. The argument type is specified by the method argument list.

StaticMethodName - Name of static Java method

string

Name of static Java method, specified as a string.

Data Types

`char`

ClassName - Name of Java class
string

Name of Java class, specified as a string, containing `StaticMethodName`.

Data Types
`char`

Tips

- In most cases, use either MATLAB or Java syntax to call methods on Java objects:

```
%MATLAB syntax
method(object,arg1,...,argn)

% Java syntax
object.method(arg1,...,argn)
```

- Use `javaMethod` to call methods having names that exceed the maximum length of a MATLAB identifier. (Call the `namelengthmax` function to obtain the maximum identifier length.)

  This is the only way you can call such a method in MATLAB. For example:

```
javaMethod('DataDefinitionAndDataManipulationTransactions',T);
```

- Use `javaMethod` when you want to specify the method name as a variable, to be invoked at runtime. When calling a static method, you also can use a variable in place of the class name argument. For example, see “Call Method Specified at Runtime” on page 1-2942.

Examples

Call Method on Java Object

Create a `java.util.Date` object, `myDate`, and change the month to `3`.

```
myDate = java.util.Date;
javaMethod('setMonth',myDate,3);
```

Call Static Method

Call `java.lang.Double` static method, `isNaN`, to test variable `num`.

```
num = 2.2;
if javaMethod('isNaN','java.lang.Double',num)
    disp('This is not a number')
end
```

Since `num` contains a number, no message is displayed.

Call Method Specified at Runtime

This example, searching for a text pattern in a string, uses variables for the pattern and for the search method. These variables could be set at runtime from user input.

Chose method, `startsWith`, and identify pattern, `str`.

```
fnc = 'startsWith';
str = java.lang.String('Four score');
```

Identify text to search.

```
gAddress = java.lang.String('Four score and seven years ago');
```

Search `gAddress`.

```
javaMethod(fnc,gAddress,str)
```

```
ans =
     1
```

`gAddress` starts with the words `Four score`.

See Also

`javaArray` | `javaObject` | `import` | `methods` | `isjava` | `javaMethodEDT`

Purpose

Call Java method from Event Dispatch Thread (EDT)

Syntax

```
javaMethodEDT(MethodName,JavaObj,x1,...,xN)
javaMethodEDT(StaticMethodName,ClassName,x1,...,xN)
```

Description

`javaMethodEDT(MethodName,JavaObj,x1,...,xN)` calls the method in the class of the Java object array with the signature matching the arguments `x1,...,xN` from the Event Dispatch Thread (EDT)

`javaMethodEDT(StaticMethodName,ClassName,x1,...,xN)` calls the static method in class `ClassName`.

Input Arguments

MethodName - Name of nonstatic Java method

string

Name of nonstatic Java method, specified as a string.

Data Types

`char`

JavaObj - Array

Java object

Array, specified as a Java object of the class containing the method.

x1,...,xN - Java method input arguments

any type

Java method input arguments, 1 through N (if any), required by `MethodName` or `StaticMethodName`, specified by any type. The argument type is specified by the method argument list.

StaticMethodName - Name of static Java method

string

Name of static Java method, specified as a string.

Data Types

`char`

ClassName - Name of Java class
string

Name of Java class, specified as a string, containing `StaticMethodName`.

Data Types
`char`

Definitions

EDT

The EDT is the Event Dispatch Thread, used in Java.

Examples

Call Method from EDT

Create a `java.util.Vector` object, `v`, and add a `string` element.

```
v = java.util.Vector;
javaMethodEDT('add',v,'string');
```

See Also

`javaMethod` | `javaObjectEDT` | `import` | `methods` | `isjava`

Purpose

Call Java constructor

Syntax

```
JavaObj = javaObject(ClassName,x1,...,xN)
```

Description

`JavaObj = javaObject(ClassName,x1,...,xN)` calls the Java constructor for the class with the argument list matching `x1,...,xN`, to create object, `JavaObj`.

Input Arguments

ClassName - Name of Java class

string

Name of Java class, specified as a string.

Data Types
`char`

x1,...,xN - Java constructor input arguments

any type

Java constructor input arguments, 1 through N (if any), required by `ClassName`, specified by any type. The argument type is specified by the class constructor argument list.

Output Arguments

JavaObj - Java object array

Java object.

Examples

Create Java Object

Create a Java object, `strObj`, of class `java.lang.String`.

```
strObj = javaObject('java.lang.String','hello');
```

See Also

`javaArray` | `javaMethod` | `import` | `methods` | `javaObjectEDT`

Concepts

- “Using the javaObjectEDT Function”

javaObjectEDT

Purpose

Call Java constructor on Event Dispatch Thread (EDT)

Syntax

```
JavaObj = javaObjectEDT(ClassName,x1,...,xN)
```

Description

`JavaObj = javaObjectEDT(ClassName,x1,...,xN)` calls the Java constructor for the class with the signature matching the arguments `x1,...,xN`, from the EDT to create object `JavaObj`.

Input Arguments

ClassName - Name of Java class

string

Name of Java class, specified as a string.

Data Types

`char`

x1,...,xN - Java constructor input arguments

any type

Java constructor input arguments, 1 through N (if any), required by `ClassName`, specified by any type. The argument type is specified by the class constructor argument list.

Output Arguments

JavaObj - Java object array

Java object.

Tips

- MATLAB calls methods on `JavaObj` from the EDT.
- Static methods on the specified class or Java object run on the MATLAB thread unless called using the `javaMethodEDT` function.

Definitions

EDT

The EDT is the Event Dispatch Thread, used in Java.

Examples

Construct Java Object Array from the EDT

```
f = javaObjectEDT('javax.swing.JFrame','New Title');
```

Call Method on Java Object

Create a `JOptionPane` on the EDT.

```
optPane = javaObjectEDT('javax.swing.JOptionPane');
```

Call the `createDialog` method on the EDT.

```
dlg = optPane.createDialog([],'Sample Dialog');
```

See Also

`javaMethodEDT` | `javaObject` | `import` | `methods`

javarmpath

Purpose

Remove entries from dynamic Java class path

Syntax

```
javarmpath(dpath1,...,dpathN)
```

Description

`javarmpath(dpath1,...,dpathN)` removes one or more files or folders from the current dynamic class path.

Input Arguments

dpath1,...,dpathN - Folders or JAR files
string

Folders or JAR files, specified as strings, to remove from path.

Data Types
`char`

Tips

- MATLAB calls the `clear java` command whenever you change the dynamic path. This command clears the definitions of all Java classes defined by files on the dynamic class path, removes all variables from the base workspace, and removes all compiled scripts, functions, and MEX-functions from memory.

Examples

Remove Folder from Dynamic Path

In order to preserve the state of the dynamic path on your system, this example first adds folders to the path, then removes one of these folders.

Create a variable that points to the MATLAB examples folder.

```
expath = fullfile(matlabroot,'extern','examples')
```

```
expath =

C:\Program Files\MATLAB\R2012b\extern\examples
```

The path reflects the folder to your MATLAB installation.

Add two folders to the path.

```
javaclasspath({...
```

```
   expath,...
   'http://www.example.com'})
javaclasspath('-dynamic')

  DYNAMIC JAVA PATH

 C:\Program Files\MATLAB\R2012b\extern\examples
 http://www.example.com
```

The output displays these new folders on your existing path.

Remove one folder.

```
javarmpath(expath)
javaclasspath('-dynamic')

  DYNAMIC JAVA PATH

 http://www.example.com
```

The path no longer contains the examples folder.

See Also

`javaclasspath` `javaaddpath` `clear`

Concepts

- “Bringing Java Classes into MATLAB Workspace”

keyboard

Purpose

Input from keyboard

Syntax

```
keyboard
```

Description

`keyboard` , when placed in a program `.m` file, stops execution of the file and gives control to the keyboard. The special status is indicated by a `K` appearing before the prompt. You can examine or change variables; all MATLAB commands are valid. This keyboard mode is useful for debugging your functions.

To terminate the keyboard mode, type `return`, and then press **Enter**. To terminate keyboard mode and exit the function, type `dbquit`, and press **Enter**.

See Also

`dbstop` | `input` | `quit` | `pause` | `return`

Purpose

Identify keys of `containers.Map` object

Syntax

```
keySet = keys(mapObj)
```

Description

`keySet = keys(mapObj)` returns cell array `keySet`, which contains all of the keys in `mapObj`.

Input Arguments

mapObj

Object of class `containers.Map`.

Output Arguments

keySet

1-by-`n` cell array, where `n` is the number of keys in `mapObj`.

Examples

Get the Keys in a Map

Create a map, and view the keys in the map:

```
myKeys = {'a','b','c'};
myValues = [1,2,3];
mapObj = containers.Map(myKeys,myValues);

keySet = keys(mapObj)
```

This code returns 1-by-3 cell array `keySet`:

```
keySet =
    'a'    'b'    'c'
```

See Also

`containers.Map` | `isKey` | `values` | `remove`

Purpose Kronecker tensor product

Syntax

```
K = kron(X,Y)
```

Description `K = kron(X,Y)` returns the Kronecker tensor product of `X` and `Y`. The result is a large array formed by taking all possible products between the elements of `X` and those of `Y`. If `X` is `m`-by-`n` and `Y` is `p`-by-`q`, then `kron(X,Y)` is `m*p`-by-`n*q`.

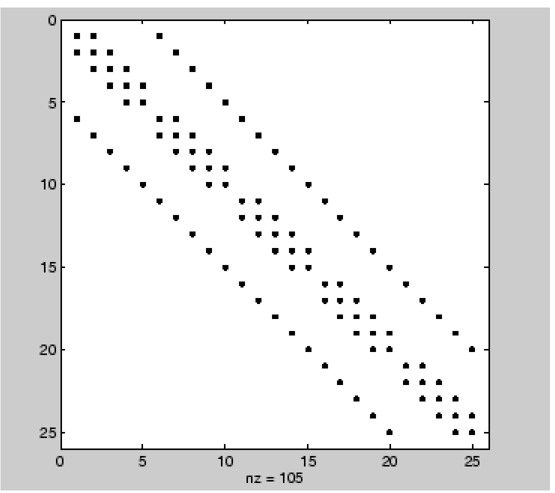
Examples If `X` is 2-by-3, then `kron(X,Y)` is

```
[ X(1,1)*Y X(1,2)*Y X(1,3)*Y
  X(2,1)*Y X(2,2)*Y X(2,3)*Y ]
```

The matrix representation of the discrete Laplacian operator on a two-dimensional, `n`-by-`n` grid is a `n^2`-by-`n^2` sparse matrix. There are at most five nonzero elements in each row or column. The matrix can be generated as the Kronecker product of one-dimensional difference operators with these statements:

```
I = speye(n,n);
E = sparse(2:n,1:n-1,1,n,n);
D = E+E'-2*I;
A = kron(D,I)+kron(I,D);
```

Plotting this with the `spy` function for `n = 5` yields:



See Also

`hankel` | `toeplitz`

last (MException)

Purpose

Last uncaught exception

Syntax

```
exception = MException.last
MException.last('reset')
```

Description

`exception = MException.last` displays the contents of the `MException` object representing your most recent uncaught error. This is a static method of the `MException` class; it is not a method of an `MException` instance.

Note Use this method from the MATLAB command line only, and not within a function.

`MException.last('reset')` sets the `identifier` and `message` properties of the exception retreived by `MException.last` to the empty string, the `stack` property to a 0-by-1 structure, and `cause` property to an empty cell array.

`MException.last` is not set if an exception is caught by a `try-catch` statement.

Examples

This example displays the last error that was caught during this MATLAB session:

```
A = 25;
A(2)
??? Index exceeds matrix dimensions.

MException.last
ans =

 MException object with properties:

    identifier: 'MATLAB:badsubscript'
       message: 'Index exceeds matrix dimensions.'
```

```
stack: [0x1 struct]
cause: {}
```

See Also

try | catch | error | assert | MException | throw(MException) | rethrow(MException) | throwAsCaller(MException) | addCause(MException) | getReport(MException)

Tiff.lastDirectory

Purpose Determine if current IFD is last in file

Syntax

```
bool = tiffobj.lastDirectory()
```

Description `bool = tiffobj.lastDirectory()` returns `true` if the current image file directory (IFD) is the last IFD in the TIFF file; otherwise, `false`.

Examples Open a `Tiff` object and determine if the current directory is the last directory in the file. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path. If the file contains only one image, the current IFD will be the last:

```
t = Tiff('myfile.tif', 'r');
tf = t.lastDirectory();
```

References This method corresponds to the `TIFFLastDirectory` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also `Tiff.setDirectory`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

Purpose

Last error message

Note `lasterr` will be removed in a future version. You can obtain information about any error that has been generated by catching an `MException`. See "Capture Information About Exceptions" in the Programming Fundamentals documentation.

Syntax

```
msgstr = lasterr
[msgstr, msgid] = lasterr
lasterr('new_msgstr')
lasterr('new_msgstr', 'new_msgid')
[msgstr, msgid] = lasterr('new_msgstr', 'new_msgid')
```

Description

`msgstr = lasterr` returns the last error message generated by the MATLAB software.

`[msgstr, msgid] = lasterr` returns the last error in `msgstr` and its message identifier in `msgid`. If the error was not defined with an identifier, `lasterr` returns an empty string for `msgid`. See "Message Identifiers" in the MATLAB Programming Fundamentals documentation for more information on the `msgid` argument and how to use it.

`lasterr('new_msgstr')` sets the last error message to a new string, `new_msgstr`, so that subsequent invocations of `lasterr` return the new error message string. You can also set the last error to an empty string with `lasterr('')`.

`lasterr('new_msgstr', 'new_msgid')` sets the last error message and its identifier to new strings `new_msgstr` and `new_msgid`, respectively. Subsequent invocations of `lasterr` return the new error message and message identifier.

`[msgstr, msgid] = lasterr('new_msgstr', 'new_msgid')` returns the last error message and its identifier, also changing these values so that subsequent invocations of `lasterr` return the message and identifier strings specified by `new_msgstr` and `new_msgid` respectively.

Examples

Example 1

Here is a function that examines the `lasterr` string and displays its own message based on the error that last occurred. This example deals with two cases, each of which is an error that can result from a matrix multiply:

```
function matrix_multiply(A, B)
try
   A * B
catch
   errmsg = lasterr;
   if(strfind(errmsg, 'Inner matrix dimensions'))
      disp('** Wrong dimensions for matrix multiply')
   else
      if(strfind(errmsg, 'not defined for variables of class'))
         disp('** Both arguments must be double matrices')
      end
   end
end
```

If you call this function with matrices that are incompatible for matrix multiplication (e.g., the column dimension of `A` is not equal to the row dimension of `B`), MATLAB catches the error and uses `lasterr` to determine its source:

```
A = [1  2  3; 6  7  2; 0  -1  5];
B = [9  5  6; 0  4  9];

matrix_multiply(A, B)
** Wrong dimensions for matrix multiply
```

Example 2

Specify a message identifier and error message string with `error`:

```
error('MyToolbox:angleTooLarge', ...
      'The angle specified must be less than 90 degrees.');
```

In your error handling code, use `lasterr` to determine the message identifier and error message string for the failing operation:

```
[errmsg, msgid] = lasterr
errmsg =
   The angle specified must be less than 90 degrees.
msgid =
   MyToolbox:angleTooLarge
```

See Also

`error` | `lasterror` | `rethrow` | `warning` | `lastwarn`

lasterror

Purpose

Last error message and related information

Note `lasterror` will be removed in a future version. You can obtain information about any error that has been generated by catching an `MException`. See "Capture Information About Exceptions" in the Programming Fundamentals documentation.

Syntax

```
s = lasterror
s = lasterror(err)
s = lasterror('reset')
```

Description

`s = lasterror` returns a structure `s` containing information about the most recent error issued by the MATLAB software. The return structure contains the following fields:

Fieldname	Description
`message`	Character array containing the text of the error message.
`identifier`	Character array containing the message identifier of the error message. If the last error issued by MATLAB had no message identifier, then the `identifier` field is an empty character array.
`stack`	Structure providing information on the location of the error. The structure has fields `file`, `name`, and `line`, and is the same as the structure returned by the `dbstack` function. If `lasterror` returns no stack information, `stack` is a 0-by-1 structure having the same three fields.

Note The `lasterror` return structure might contain additional fields in future versions of MATLAB.

The fields of the structure returned in `stack` are

Fieldname	Description
`file`	Name of the file in which the function generating the error appears. This field is the empty string if there is no file.
`name`	Name of the function in which the error occurred. If this is the primary function in the file, and the function name differs from the file name, `name` is set to the file name.
`line`	Line number of the file at which the error occurred.

See "Message Identifiers" in the MATLAB Programming Fundamentals documentation for more information on the syntax and usage of message identifiers.

`s = lasterror(err)` sets the last error information to the error message and identifier specified in the structure `err`. Subsequent invocations of `lasterror` return this new error information. The optional return structure `s` contains information on the previous error.

`s = lasterror('reset')` sets the last error information to the default state. In this state, the `message` and `identifier` fields of the return structure are empty strings, and the `stack` field is a 0-by-1 structure.

Tips

MathWorks is gradually transitioning MATLAB error handling to an object-oriented scheme that is based on the `MException` class. Although support for `lasterror` is expected to continue, using the static `last` method of `MException` is preferable.

Warning

`lasterror` and `MException.last` are not guaranteed to always return identical results. For example, `MException.last` updates its error status only on uncaught errors, where `lasterror` can update its error status on any error, whether it is caught or not.

Examples

Example 1

Save the following MATLAB code in a file called `average.m`:

```
function y = average(x)
% AVERAGE Mean of vector elements.
% AVERAGE(X), where X is a vector, is the mean of vector elements.
% Nonvector input results in an error.
check_inputs(x)
y = sum(x)/length(x);      % The actual computation

function check_inputs(x)
[m,n] = size(x);
if (~((m == 1) || (n == 1)) || (m == 1 && n == 1))
    error('AVG:NotAVector', 'Input must be a vector.')
end
```

Now run the function. Because this function requires vector input, passing a scalar value to it forces an error. The error occurs in subroutine `check_inputs`:

```
average(200)
Error using average>check_inputs (line 11)
Input must be a vector.

Error in average (line 5)
check_inputs(x)
```

Get the three fields from `lasterror`:

```
err = lasterror
err =
       message: [1x61 char]
    identifier: 'AVG:NotAVector'
         stack: [2x1 struct]
```

Display the text of the error message:

```
msg = err.message
```

```
msg =
    Error using average>check_inputs (line 11)
    Input must be a vector.
```

Display the fields containing the `stack` information. `err.stack` is a 2-by-1 structure because it provides information on the failing subroutine `check_inputs` and also the outer, primary function `average`:

```
st1 = err.stack(1,1)
st1 =
    file: 'd:\matlab_test\average.m'
    name: 'check_inputs'
    line: 11

st2 = err.stack(2,1)
st2 =
    file: 'd:\matlab_test\average.m'
    name: 'average'
    line: 5
```

Note As a rule, the name of your primary function should be the same as the name of the file that contains that function. If these names differ, MATLAB uses the file name in the `name` field of the `stack` structure.

Example 2

`lasterror` is often used in conjunction with the `rethrow` function in `try-catch` statements. For example,

```
try
   do_something
catch
   do_cleanup
   rethrow(lasterror)
end
```

See Also

`last(MException)` | `MException` | `try` | `catch` | `error` | `assert` | `rethrow` | `lastwarn` | `dbstack`

Purpose

Last warning message

Syntax

```
msgstr = lastwarn
[msgstr, msgid] = lastwarn
lastwarn('new_msgstr')
lastwarn('new_msgstr', 'new_msgid')
[msgstr, msgid] = lastwarn('new_msgstr', 'new_msgid')
```

Description

`msgstr = lastwarn` returns the last warning message generated by the MATLAB software.

`[msgstr, msgid] = lastwarn` returns the last warning in `msgstr` and its message identifier in `msgid`. If the warning was not defined with an identifier, `lastwarn` returns an empty string for `msgid`.

`lastwarn('new_msgstr')` sets the last warning message to a new string, `new_msgstr`, so that subsequent invocations of `lastwarn` return the new warning message string. You can also set the last warning to an empty string with `lastwarn('')`.

`lastwarn('new_msgstr', 'new_msgid')` sets the last warning message and its identifier to new strings `new_msgstr` and `new_msgid`, respectively. Subsequent invocations of `lastwarn` return the new warning message and message identifier.

`[msgstr, msgid] = lastwarn('new_msgstr', 'new_msgid')` returns the last warning message and its identifier, also changing these values so that subsequent invocations of `lastwarn` return the message and identifier strings specified by `new_msgstr` and `new_msgid`, respectively.

Tips

`lastwarn` does not return warnings that are reported during the parsing of MATLAB commands. (Warning messages that include the failing file name and line number are parse-time warnings.)

Examples

Write a short function that generates a warning message. At the start of the function, enable any warnings that have a message identifier called `TestEnv:InvalidInput`:

```
function myfun(p1)
warning on TestEnv:InvalidInput;

exceedMax = find(p1 > 5000);
if any(exceedMax)
    warning('TestEnv:InvalidInput', ...
        '%d values in the "%s" array exceed the maximum.', ...
        length(exceedMax), inputname(1))
end
```

Pass an invalid value to the function:

```
dataIn = magic(10) - 2;

myfun(dataIn)
Warning: 2 values in the "dataIn" array exceed the maximum.
> In myfun at 4
```

Use `lastwarn` to determine the message identifier and error message string for the operation:

```
[warnmsg, msgid] = lastwarn
warnmsg =
   2 values in the "dataIn" array exceed the maximum.
msgid =
   TestEnv:InvalidInput
```

See Also

`warning` | `error`

Related Examples

- "Issue Warnings and Errors"
- "Suppress Warnings"
- "Restore Warnings"

Purpose

Least common multiple

Syntax

```
L = lcm(A,B)
```

Description

`L = lcm(A,B)` returns the least common multiples of the elements of `A` and `B`.

Input Arguments

A,B - Input values

scalars, vectors, or arrays of real, positive integer values

Input values, specified as scalars, vectors, or arrays of real, positive integer values. `A` and `B` can be any numeric type, and they can be of different types within certain limitations:

- If `A` or `B` is of type `single`, then the other can be of type `single` or `double`.
- If `A` or `B` belongs to an integer class, then the other must belong to the same class or it must be a `double` scalar value.

`A` and `B` must be the same size or one must be a scalar.

Example: `[20 3 13],[10 6 7]`

Example: `int16([100 30 200]),int16([20 15 9])`

Example: `int16([100 30 200]),20`

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

L - Least common multiple

real, positive integer values

Least common multiple, returned as an array of real positive integer values. `L` is the same size as `A` and `B`, and it has the same type as `A` and `B`. If `A` and `B` are of different types, then `L` is returned as the nondouble type.

Examples

Least Common Multiples of Double Array and a Scalar

```
A = [5 17; 10 60];
B = 45;
L = lcm(A,B)

L =

    45   765
    90   180
```

Least Common Multiples of Unsigned Integers

```
A = uint16([255 511 15]);
B = uint16([15 127 1023]);
L = lcm(A,B)

L =

    255  64897   5115
```

See Also

`gcd`

Purpose

Block LDL' factorization for Hermitian indefinite matrices

Syntax

```
L = ldl(A)
[L,D] = ldl(A)
[L,D,P] = ldl(A)
[L,D,p] = ldl(A,'vector')
[U,D,P] = ldl(A,'upper')
[U,D,p] = ldl(A,'upper','vector')
[L,D,P,S] = ldl(A)
[L,D,P,S] = LDL(A,THRESH)
[U,D,p,S] = LDL(A,THRESH,'upper','vector')
```

Description

`L = ldl(A)` returns only the "psychologically lower triangular matrix" `L` as in the two-output form. The permutation information is lost, as is the block diagonal factor `D`. By default, `ldl` references only the diagonal and lower triangle of `A`, and assumes that the upper triangle is the complex conjugate transpose of the lower triangle. Therefore `[L,D,P] = ldl(TRIL(A))` and `[L,D,P] = ldl(A)`both return the exact same factors. Note, this syntax is not valid for sparse `A`.

`[L,D] = ldl(A)` stores a block diagonal matrix `D` and a "psychologically lower triangular matrix" (i.e a product of unit lower triangular and permutation matrices) in `L` such that `A = L*D*L'`. The block diagonal matrix `D` has 1-by-1 and 2-by-2 blocks on its diagonal. Note, this syntax is not valid for sparse `A`.

`[L,D,P] = ldl(A)` returns unit lower triangular matrix `L`, block diagonal `D`, and permutation matrix `P` such that `P'*A*P = L*D*L'`. This is equivalent to `[L,D,P] = ldl(A,'matrix')`.

`[L,D,p] = ldl(A,'vector')` returns the permutation information as a vector, `p`, instead of a matrix. The `p` output is a row vector such that `A(p,p) = L*D*L'`.

`[U,D,P] = ldl(A,'upper')` references only the diagonal and upper triangle of `A` and assumes that the lower triangle is the complex conjugate transpose of the upper triangle. This syntax returns a unit upper triangular matrix `U` such that `P'*A*P = U'*D*U` (assuming that

`A` is Hermitian, and not just upper triangular). Similarly, `[L,D,P] = ldl(A,'lower')` gives the default behavior.

`[U,D,p] = ldl(A,'upper','vector')` returns the permutation information as a vector, `p`, as does `[L,D,p] = ldl(A,'lower','vector')`. `A` must be a full matrix.

`[L,D,P,S] = ldl(A)` returns unit lower triangular matrix `L`, block diagonal `D`, permutation matrix `P`, and scaling matrix `S` such that `P'*S*A*S*P = L*D*L'`. This syntax is only available for real sparse matrices, and only the lower triangle of `A` is referenced. `ldl` uses MA57 for sparse real symmetric `A`.

`[L,D,P,S] = LDL(A,THRESH)` uses `THRESH` as the pivot tolerance in MA57. `THRESH` must be a double scalar lying in the interval `[0, 0.5]`. The default value for `THRESH` is `0.01`. Using smaller values of `THRESH` may give faster factorization times and fewer entries, but may also result in a less stable factorization. This syntax is available only for real sparse matrices.

`[U,D,p,S] = LDL(A,THRESH,'upper','vector')` sets the pivot tolerance and returns upper triangular `U` and permutation vector `p` as described above.

Examples

These examples illustrate the use of the various forms of the `ldl` function, including the one-, two-, and three-output form, and the use of the `vector` and `upper` options. The topics covered are:

- "Example 1 — Two-Output Form of ldl" on page 1-2971
- "Example 2 — Three Output Form of ldl" on page 1-2971
- "Example 3 — The Structure of D" on page 1-2972
- "Example 4 — Using the 'vector' Option" on page 1-2972
- "Example 5 — Using the 'upper' Option" on page 1-2973
- "Example 6 — linsolve and the Hermitian indefinite solver" on page 1-2973

Before running any of these examples, you will need to generate the following positive definite and indefinite Hermitian matrices:

```
A = full(delsq(numgrid('L', 10)));
B = gallery('uniformdata',10,0);
M = [eye(10) B; B' zeros(10)];
```

The structure of M here is very common in optimization and fluid-flow problems, and M is in fact indefinite. Note that the positive definite matrix A must be full, as ldl does not accept sparse arguments.

Example 1 — Two-Output Form of ldl

The two-output form of ldl returns L and D such that A-(L*D*L') is small, L is "psychologically unit lower triangular" (i.e., a permuted unit lower triangular matrix), and D is a block 2-by-2 diagonal. Note also that, because A is positive definite, the diagonal of D is all positive:

```
[LA,DA] = ldl(A);
fprintf(1, ...
'The factorization error ||A - LA*DA*LA''|| is %g\n', ...
norm(A - LA*DA*LA'));
neginds = find(diag(DA) < 0)
```

Given a b, solve Ax=b using LA, DA:

```
bA = sum(A,2);
x = LA'\(DA\(LA\bA));
fprintf(...
'The absolute error norm ||x - ones(size(bA))|| is %g\n', ...
norm(x - ones(size(bA))));
```

Example 2 — Three Output Form of ldl

The three output form returns the permutation matrix as well, so that L is in fact unit lower triangular:

```
[Lm, Dm, Pm] = ldl(M);
fprintf(1, ...
'The error norm ||Pm''*M*Pm - Lm*Dm*Lm''|| is %g\n', ...
```

```
norm(Pm'*M*Pm - Lm*Dm*Lm'));
fprintf(1, ...
'The difference between Lm and tril(Lm) is %g\n', ...
norm(Lm - tril(Lm)));
```

Given b, solve Mx=b using Lm, Dm, and Pm:

```
bM = sum(M,2);
x = Pm*(Lm'\(Dm\(Lm\(Pm'*bM))));
fprintf(...
'The absolute error norm ||x - ones(size(b))|| is %g\n', ...
norm(x - ones(size(bM))));
```

Example 3 — The Structure of D

D is a block diagonal matrix with 1-by-1 blocks and 2-by-2 blocks. That makes it a special case of a tridiagonal matrix. When the input matrix is positive definite, D is almost always diagonal (depending on how definite the matrix is). When the matrix is indefinite however, D may be diagonal or it may express the block structure. For example, with A as above, DA is diagonal. But if you shift A just a bit, you end up with an indefinite matrix, and then you can compute a D that has the block structure.

```
figure; spy(DA); title('Structure of D from ldl(A)');
[Las, Das] = ldl(A - 4*eye(size(A)));
figure; spy(Das);
title('Structure of D from ldl(A - 4*eye(size(A)))');
```

Example 4 — Using the 'vector' Option

Like the lu function, ldl accepts an argument that determines whether the function returns a permutation vector or permutation matrix. ldl returns the latter by default. When you select 'vector', the function executes faster and uses less memory. For this reason, specifying the 'vector' option is recommended. Another thing to note is that indexing is typically faster than multiplying for this kind of operation:

```
[Lm, Dm, pm] = ldl(M, 'vector');
```

```
fprintf(1, 'The error norm ||M(pm,pm) - Lm*Dm*Lm''|| is %g\n', ...
  norm(M(pm,pm) - Lm*Dm*Lm'));

% Solve a system with this kind of factorization.
clear x;
x(pm,:) = Lm'\(Dm\(Lm\(bM(pm,:))));
fprintf('The absolute error norm ||x - ones(size(b))|| is %g\n', ...
  norm(x - ones(size(bM))));
```

Example 5 — Using the 'upper' Option

Like the `chol` function, `ldl` accepts an argument that determines which triangle of the input matrix is referenced, and also whether `ldl` returns a lower (`L`) or upper (`L'`) triangular factor. For dense matrices, there are no real savings with using the upper triangular version instead of the lower triangular version:

```
Ml = tril(M);
[Lml, Dml, Pml] = ldl(Ml, 'lower'); % 'lower' is default behavior.
fprintf(1, ...
'The difference between Lml and Lm is %g\n', norm(Lml - Lm));
[Umu, Dmu, pmu] = ldl(triu(M), 'upper', 'vector');
fprintf(1, ...
'The difference between Umu and Lm'' is %g\n', norm(Umu - Lm'));

% Solve a system using this factorization.
clear x;
x(pm,:) = Umu\(Dmu\(Umu'\(bM(pmu,:))));
fprintf(...
'The absolute error norm ||x - ones(size(b))|| is %g\n', ...
norm(x - ones(size(bM))));
```

When specifying both the `'upper'` and `'vector'` options, `'upper'` must precede `'vector'` in the argument list.

Example 6 — linsolve and the Hermitian indefinite solver

When using the `linsolve` function, you may experience better performance by exploiting the knowledge that a system has a symmetric

matrix. The matrices used in the examples above are a bit small to see this so, for this example, generate a larger matrix. The matrix here is symmetric positive definite, and below we will see that with each bit of knowledge about the matrix, there is a corresponding speedup. That is, the symmetric solver is faster than the general solver while the symmetric positive definite solver is faster than the symmetric solver:

```
Abig = full(delsq(numgrid('L', 30)));
bbig = sum(Abig, 2);
LSopts.POSDEF = false;
LSopts.SYM = false;
tic; linsolve(Abig, bbig, LSopts); toc;
LSopts.SYM = true;
tic; linsolve(Abig, bbig, LSopts); toc;
LSopts.POSDEF = true;
tic; linsolve(Abig, bbig, LSopts); toc;
```

Algorithms

`ldl` uses the MA57 routines in the Harwell Subroutine Library (HSL) for real sparse matrices.

References

[1] Ashcraft, C., R.G. Grimes, and J.G. Lewis. “Accurate Symmetric Indefinite Linear Equations Solvers.” *SIAM J. Matrix Anal. Appl.* Vol. 20. Number 2, 1998, pp. 513–561.

[2] Duff, I. S. "MA57 — A new code for the solution of sparse symmetric definite and indefinite systems." Technical Report RAL-TR-2002-024, Rutherford Appleton Laboratory, 2002.

See Also

`chol` | `lu` | `qr`

Purpose

Left array division

Syntax

```
x = B.\A
x = ldivide(B,A)
```

Description

`x = B.\A` divides each element of `A` by the corresponding element of `B`.

- If `A` and `B` are arrays, then they must be the same size.
- If either `A` or `B` is a scalar, then MATLAB expands the scalar value into an appropriately sized array.

`x = ldivide(B,A)` is an alternative way to divide `A` by `B`, but is rarely used. It enables operator overloading for classes.

Input Arguments

A - Numerator

numeric array | sparse numeric array

Numerator, specified as a full or sparse numeric array. If `B` is an integer data type, then `A` must be the same integer type or a scalar double.

Data Types

`single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

Complex Number Support: Yes

B - Denominator

numeric array | sparse numeric array

Denominator, specified as a full or sparse numeric array. If `A` is an integer data type, then `B` must be the same integer type or a scalar double.

Data Types

`single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

Complex Number Support: Yes

Output Arguments

x - Solution

numeric array | sparse numeric array

Solution, specified as a sparse or full numeric array. If either A or B are integer data types, then x is that same integer data type.

Examples

Divide Two Numeric Arrays

```
A = ones(2, 3);
B = [1 2 3;4 5 6];
x = B.\A

x =

    1.0000    0.5000    0.3333
    0.2500    0.2000    0.1667
```

Divide a Scalar by a Numeric Array

```
C = 2;
D = [1 2 3;4 5 6];
x = D.\C

x =

    2.0000    1.0000    0.6667
    0.5000    0.4000    0.3333
```

Tips

- MATLAB does not support complex integer division.

See Also

rdivide | mldivide | mrdivide | Arithmetic Operators \,/

Purpose

Test for less than or equal to

Syntax

```
A <= B
le(A, B)
```

Description

`A <= B` compares each element of array `A` with the corresponding element of array `B`, and returns an array with elements set to logical 1 (`true`) where `A` is less than or equal to `B`, or set to logical 0 (`false`) where `A` is greater than `B`. Each input of the expression can be an array or a scalar value.

If both `A` and `B` are scalar (i.e., 1-by-1 matrices), then the MATLAB software returns a scalar value.

If both `A` and `B` are nonscalar arrays, then these arrays must have the same dimensions, and MATLAB returns an array of the same dimensions as `A` and `B`.

If one input is scalar and the other a nonscalar array, then the scalar input is treated as if it were an array having the same dimensions as the nonscalar input array. In other words, if input `A` is the number 100, and `B` is a 3-by-5 matrix, then `A` is treated as if it were a 3-by-5 matrix of elements, each set to 100. MATLAB returns an array of the same dimensions as the nonscalar input array.

`le(A, B)` is called for the syntax `A < =B` when either `A` or `B` is an object.

Examples

Create two 6-by-6 matrices, `A` and `B`, and locate those elements of `A` that are less than or equal to the corresponding elements of `B`:

```
A = magic(6);
B = repmat(3*magic(3), 2, 2);

A <= B
ans =
     0     1     1     0     0     0
     1     0     1     0     0     0
     0     1     1     0     1     0
     1     0     0     1     0     1
```

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

See Also `lt` | `eq` | `ge` | `gt` | `ne` | Relational Operators

Purpose

Graph legend for lines and patches

Syntax

```
legend('string1','string2',...)
legend(h,'string1','string2',...)
legend(M)
legend(h,M)
legend(M,'parameter_name','parameter_value',...)
legend(h,M,'parameter_name','parameter_value',...)
legend(axes_handle,...)
legend('off'),legend(axes_handle,'off')
legend('toggle'),legend(axes_handle,'toggle')
legend('hide'),legend(axes_handle,'hide')
legend('show'),legend(axes_handle,'show')
legend('boxoff'),legend(axes_handle,'boxoff')
legend('boxon'),legend(axes_handle,'boxon')
legend_handle = legend(...)
legend(...,'Location','location')
legend(...,'Orientation','orientation')
[legend_h,object_h,plot_h,text_strings] = legend(...)
```

Description

The `legend` function places a legend on various types of graphs (line plots, bar graphs, pie charts, etc.). For each line plotted, the legend shows a sample of the line type, marker symbol, and color beside the text label you specify. When plotting filled areas (patch or surface objects), the legend contains a sample of the face color next to the text label.

The font size and font name for the legend strings match the axes `FontSize` and `FontName` properties.

`legend('string1','string2',...)` displays a legend in the current axes using the specified strings to label each set of data.

`legend(h,'string1','string2',...)` displays a legend on the plot containing the objects identified by the handles in the vector `h` and uses the specified strings to label the corresponding graphics object (line, barseries, etc.).

`legend(M)` adds a legend containing the rows of the matrix or cell array of strings `M` as labels. For matrices, this is the same as `legend(M(1,:),M(2,:),...)`.

`legend(h,M)` associates each row of the matrix or cell array of strings `M` with the corresponding graphics object (patch or line) in the vector of handles `h`.

`legend(M,'parameter_name','parameter_value',...)` and `legend(h,M,'parameter_name','parameter_value',...)` allow parameter/value pairs to be set when creating a legend (you can also assign them with `set` or with the Property Editor or Property Inspector). `M` must be a cell array of names. Legends inherit the properties of axes, although not all of them are relevant to legend objects.

`legend(axes_handle,...)` displays the legend for the axes specified by `axes_handle`.

`legend('off'),legend(axes_handle,'off')` removes the legend in the current axes or the axes specified by `axes_handle`.

`legend('toggle'),legend(axes_handle,'toggle')` toggles the legend on or off. If no legend exists for the current axes, one is created using default strings.

The *default string* for an object is the value of the object's `DisplayName` property, if you have defined a value for `DisplayName` (which you can do using the Property Editor or calling `set`). Otherwise, `legend` constructs a string of the form `data1`, `data2`, etc. Setting display names is useful when you are experimenting with legends and might forget how objects in a lineseries, for example, are ordered.

When you specify legend strings in a `legend` command, their respective `DisplayName`s are set to these strings. If you delete a legend and then create a new legend without specifying labels for it, the values of `DisplayName` are (re)used as label names. Naturally, the associated plot objects must have a `DisplayName` property for this to happen: all *`_series`* and *`_group`* plot objects have a `DisplayName` property; Handle Graphics primitives, such as `line` and `patch`, do not.

Legends for graphs that contain groups of objects such as lineseries, barseries, contourgroups, etc. created by high-level plotting commands such as `plot`, `bar`, `contour`, etc., by default display a single legend entry for the entire group, regardless of how many member objects it contains. However, you can customize such legends to show individual entries for all or selected member objects and assign a unique `DisplayName` to any of them. You control how groups appear in the legend by setting values for their `Annotation` and `DisplayName` properties with code. For information and examples about customizing legends in this manner, see "Controlling Legends" in the MATLAB Graphics documentation.

You can specify `EdgeColor` and `TextColor` as RGB triplets or as `ColorSpecs`. You cannot set these colors to `'none'`. To hide the box surrounding a legend, set the `Box` property to `'off'`. To allow the background to show through the legend box, set the legend's `Color` property to `'none'`, for example,

```
set(legend_handle, 'Box', 'off')
set(legend_handle, 'Color', 'none')
```

This is similar to the effect of the command `legend boxoff`, except that `boxoff` also hides the legend's border.

You can use a legend's handle to set text properties for all the strings in a legend at once with a cell array of strings, rather than looping through each of them. See the last line of the example below, which demonstrates setting a legend's `Interpreter` property. In that example, you could reset the `String` property of the legend as follows:

```
set(h,'String',{'cos(x)','sin(x)'})
```

See the documentation for Text Properties for additional details.

`legend('hide'),legend(axes_handle,'hide')` makes the legend in the current axes or the axes specified by `axes_handle` invisible.

`legend('show'),legend(axes_handle,'show')` makes the legend in the current axes or the axes specified by `axes_handle` visible. A legend is created if one did not exist previously. Legends created automatically are limited to depict only the first 20 lines in the plot; if you need more

legend entries, you can manually create a legend for them all with `legend('string1','string2',...)` syntax.

`legend('boxoff'),legend(axes_handle,'boxoff')` removes the box from the legend in the current axes or the axes specified by `axes_handle`, and makes its background transparent.

`legend('boxon'),legend(axes_handle,'boxon')` adds a box with an opaque background to the legend in the current axes or the axes specified by `axes_handle`.

You can also type the above six commands using the syntax

```
legend keyword
```

If the keyword is not recognized, it is used as legend text, creating a legend or replacing the current legend.

`legend_handle = legend(...)` returns the handle to the legend on the current axes, or `[]` if no legend exists.

`legend(...,'Location','location')` uses *`location`* to determine where to place the legend. *`location`* can be either be a 1-by-4 position vector (`[left,bottom,width,height]`) or one of the following strings.

Specifier	Location in Axes
`North`	Inside plot box near top
`South`	Inside bottom
`East`	Inside right
`West`	Inside left
`NorthEast`	Inside top right (default for 2-D plots)
`NorthWest`	Inside top left
`SouthEast`	Inside bottom right
`SouthWest`	Inside bottom left
`NorthOutside`	Outside plot box near top

Specifier	Location in Axes
`SouthOutside`	Outside bottom
`EastOutside`	Outside right
`WestOutside`	Outside left
`NorthEastOutside`	Outside top right (default for 3-D plots)
`NorthWestOutside`	Outside top left
`SouthEastOutside`	Outside bottom right
`SouthWestOutside`	Outside bottom left
`Best`	Least conflict with data in plot
`BestOutside`	Least unused space outside plot

Using one of the `...Outside` values for '*`Location`*' ensures that the legend does not overlap the plot, whereas overlaps can occur when you specify any of the other cardinal values. The *`location`* property applies to colorbars and legends, but not to axes.

Note You can set the legend location by passing the 4-element position vector to the `legend` function using the `` `Location' `` option. To define the position of an existing legend, use the `set` function to assign the 4-element position vector to the `` `Position' `` property. You cannot use the `Location` option with the `set` function.

Obsolete Location Values

The first column of the following table shows the now-obsolete specifiers for legend locations that were in use prior to Version 7, along with a description of the locations and their current equivalent syntaxes:

Obsolete Specifier	Location in Axes	Current Specifier
-1	Outside axes on right side	NorthEastOutside
0	Inside axes	Best
1	Upper right corner of axes	NorthEast
2	Upper left corner of axes	NorthWest
3	Lower left corner of axes	SouthWest
4	Lower right corner of axes	SouthEast

`legend(...,'Orientation','orientation')` creates a legend with the legend items arranged in the specified orientation. *`orientation`* can be `vertical` (the default) or `horizontal`.

`[legend_h,object_h,plot_h,text_strings] = legend(...)` returns

- `legend_h` — Handle of the legend axes
- `object_h` — Handles of the line, patch, and text graphics objects used in the legend
- `plot_h` — Handles of the lines and other objects used in the plot
- `text_strings` — Cell array of the text strings used in the legend

Relationship to Axes

`legend` associates strings with the objects in the axes in the same order that they are listed in the axes `Children` property. By default, the legend annotates the current axes.

You can only display one legend per axes. `legend` positions the legend based on a variety of factors, such as what objects the legend obscures.

The properties that legends do not share with axes are

- `Location`
- `Orientation`
- `EdgeColor`

- `TextColor`
- `Interpreter`
- `String`

Tips

Using Keywords as Legend Labels

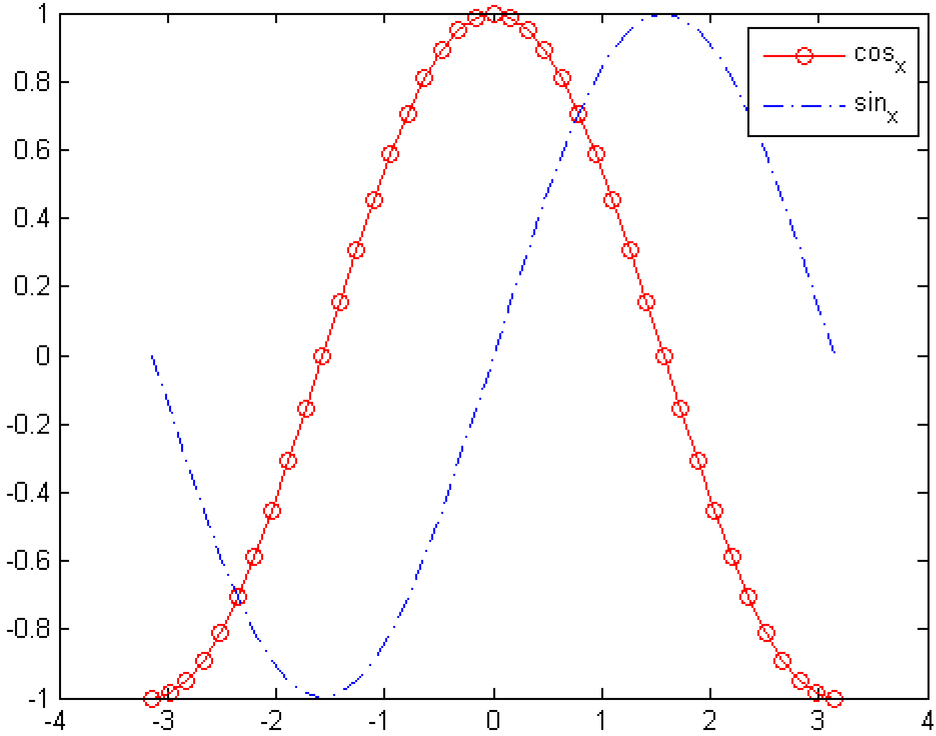
To use legend keywords like `'Location'`, `` `Orientation' ``, `` `Off' ``, `'Hide'`, or property names as legend labels, pass the string in a cell array to the `legend` function as,

```
l = legend({'Location'});
```

Examples

Add a legend to a graph showing a sine and cosine function. The default location is upper right, within the axes:

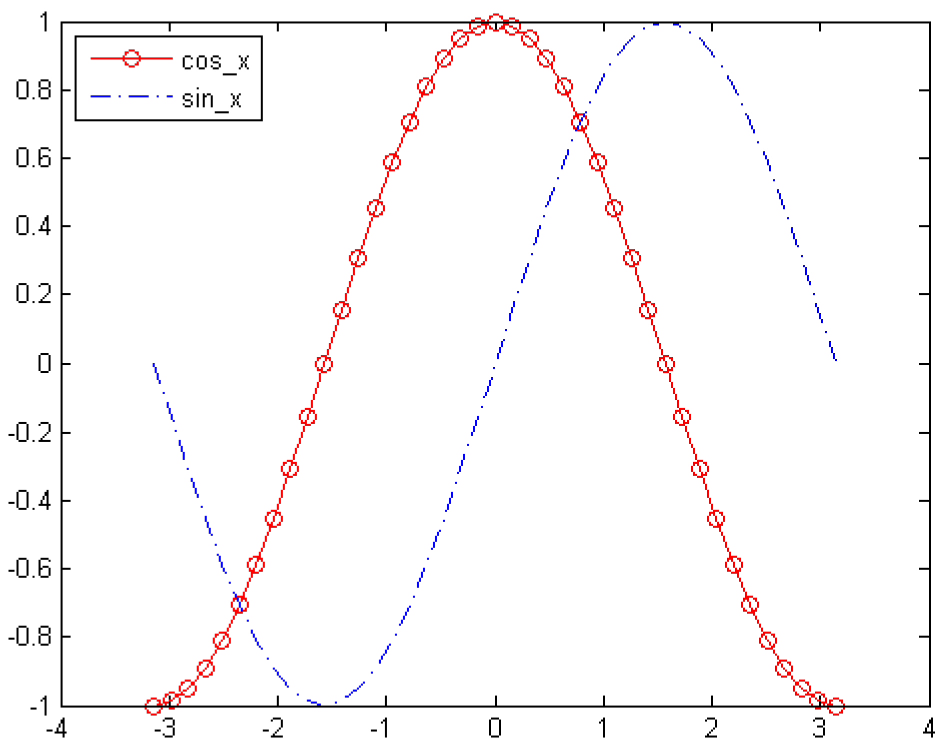
```
figure
x = -pi:pi/20:pi;
plot(x,cos(x),'-ro',x,sin(x),'-.b')
hleg1 = legend('cos_x','sin_x');
```



The legend reflects that `plot` specified a solid, red line (`'-ro'`) for the cosine function and a dash-dot, blue line (`'-.b'`) for the sine function.

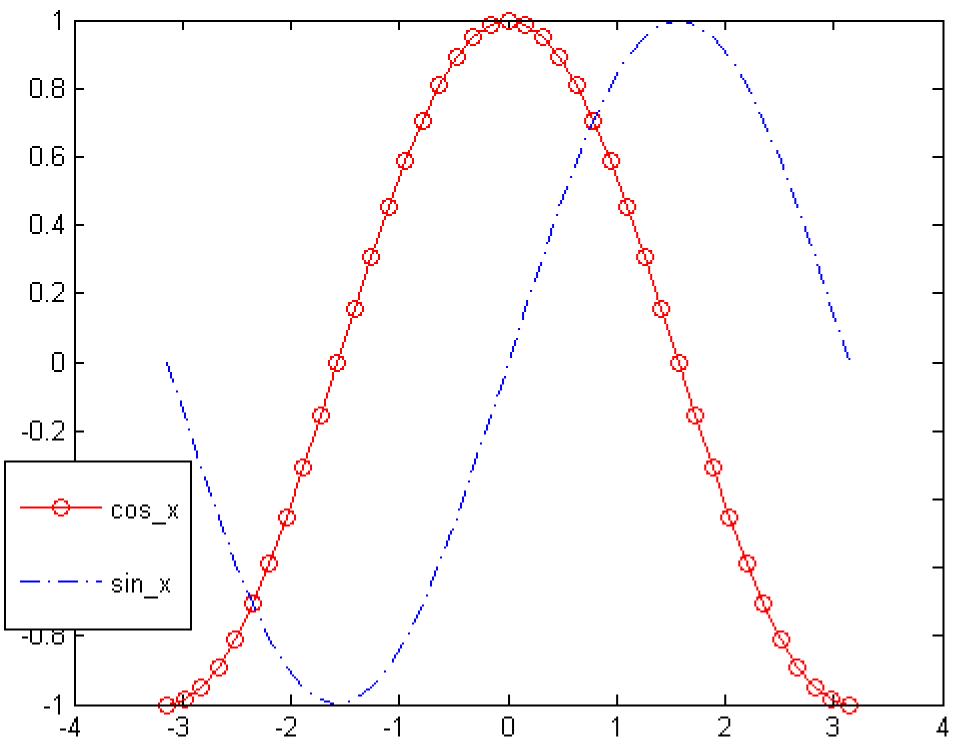
Update the legend. Use the returned legend handle, `hleg1`, to move the legend to the upper left. Also turn off the TeX interpreter to render underscores in legend text literally rather than as subscripts:

```
set(hleg1,'Location','NorthWest')
set(hleg1,'Interpreter','none')
```



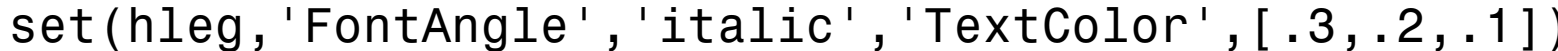
Setting the position of the legend at a particular position.

```
set(hleg1, 'Position', [.1,.2,.1,.2]);
```

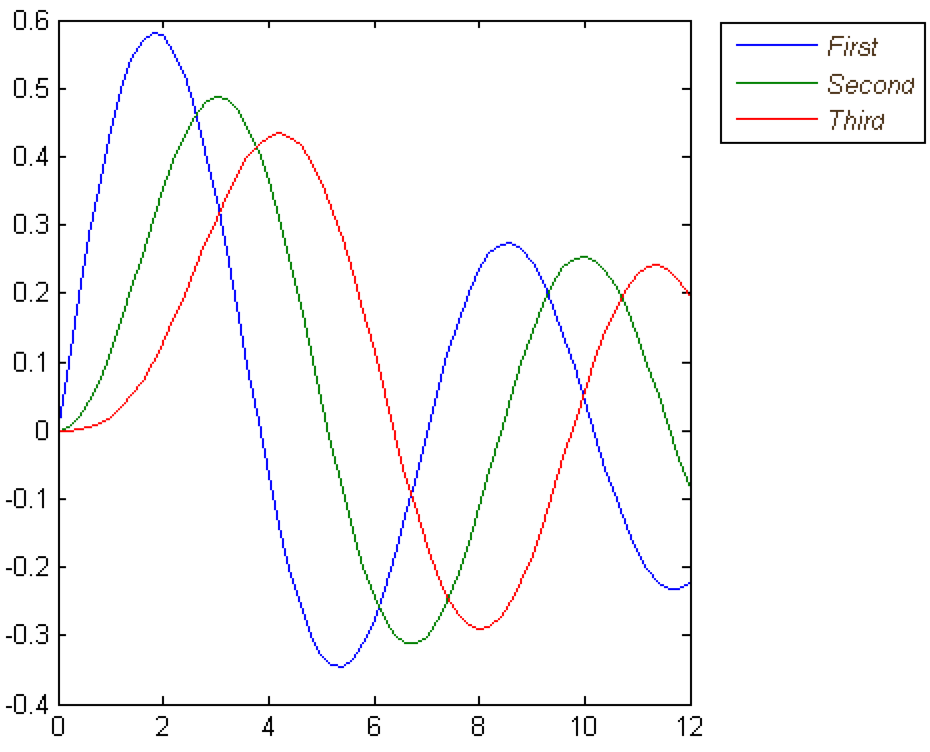


Use `besselj` to plot Bessel functions for orders 1, 2, and 3. Add a legend for the lines at the upper right, outside the axes.

```
figure
x = 0:.2:12;
plot(x,besselj(1,x),x,besselj(2,x),x,besselj(3,x));
hleg = legend('First','Second','Third',...
              'Location','NorthEastOutside')
% Make the text of the legend italic and color it brown
```



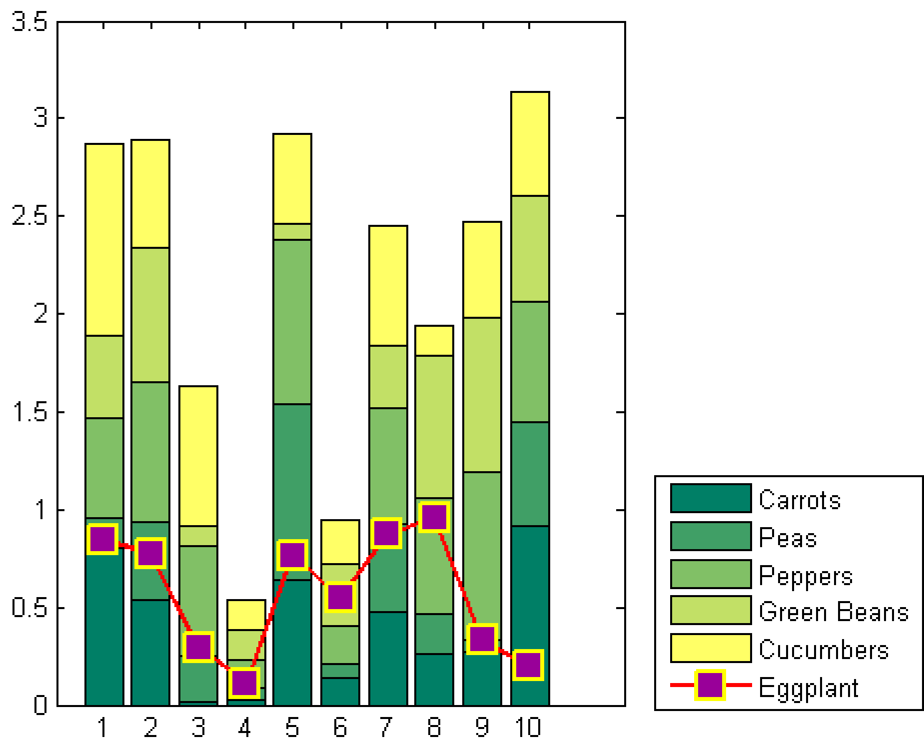
```
set(hleg,'FontAngle','italic','TextColor',[.3,.2,.1])
```



Create a bar graph and overlay a line plot on it by setting `hold on`. Create a legend that reflects both graphs and locate it to the lower right, outside the axes:

```
figure
stream = RandStream('mrg32k3a','Seed',4);
y1 = rand(stream,10,5);
```

```
hb = bar(y1,'stacked'); colormap(summer); hold on
y2 = rand(stream,10,1);
hp = plot(1:10,y2,'marker','square','markersize',12,...
       'markeredgecolor','y','markerfacecolor',[.6,0,.6],...
       'linestyle','-','color','r','linewidth',2); hold off
legend([hb,hp],'Carrots','Peas','Peppers','Green Beans',...
        'Cucumbers','Eggplant','Location','SouthEastOutside')
```



Alternatives

Add a legend to a selected axes on a graph with the **Insert Legend** tool on the figure toolbar, or use **Insert > Legend** from the figure menu. Use the Property Editor to modify the position, font, and other properties of a legend. For details, see Using Plot Edit Mode.

Moving the Legend

Move the legend by pressing the left mouse button while the cursor is over the legend and dragging the legend to a new location. Double-clicking a label allows you to edit the label.

See Also

`LineSpec` | `plot` | Text Properties

How To

- "Adding a Legend to a Graph"

Purpose

Associated Legendre functions

Syntax

```
P = legendre(n,X)
S = legendre(n,X,'sch')
N = legendre(n,X,'norm')
```

Description

`P = legendre(n,X)` computes the associated Legendre functions of degree `n` and order `m = 0,1,...,n`, evaluated for each element of `X`. Argument `n` must be a scalar integer, and `X` must contain real values in the domain $-1 \leq x \leq 1$.

If `X` is a vector, then `P` is an `(n+1)`-by-`q` matrix, where `q = length(X)`. Each element `P(m+1,i)` corresponds to the associated Legendre function of degree `n` and order `m` evaluated at `X(i)`.

In general, the returned array `P` has one more dimension than `X`, and each element `P(m+1,i,j,k,...)` contains the associated Legendre function of degree `n` and order `m` evaluated at `X(i,j,k,...)`. Note that the first row of `P` is the Legendre polynomial evaluated at `X`, i.e., the case where `m` = 0.

`S = legendre(n,X,'sch')` computes the "Schmidt Seminormalized Associated Legendre Functions" on page 1-2993.

`N = legendre(n,X,'norm')` computes the "Fully Normalized Associated Legendre Functions" on page 1-2993.

Definitions

Associated Legendre Functions

The Legendre functions are defined by

$$P_n^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_n(x),$$

where

$$P_n(x)$$

is the Legendre polynomial of degree n:

$$P_n(x) = \frac{1}{2^n n!}\left[\frac{d^n}{dx^n}(x^2-1)^n\right]$$

Schmidt Seminormalized Associated Legendre Functions

The Schmidt seminormalized associated Legendre functions are related to the nonnormalized associated Legendre functions $P_n^m(x)$ by

$$P_n(x) \text{ for } m = 0,$$

$$S_n^m(x) = (-1)^m \sqrt{\frac{2(n-m)!}{(n+m)!}} P_n^m(x) \text{ for } m > 0.$$

Fully Normalized Associated Legendre Functions

The fully normalized associated Legendre functions are normalized such that

$$\int_{-1}^{1} \left(N_n^m(x)\right)^2 dx = 1$$

and are related to the unnormalized associated Legendre functions $P_n^m(x)$ by

$$N_n^m = (-1)^m \sqrt{\frac{(n+\frac{1}{2})(n-m)!}{(n+m)!}} P_n^m(x)$$

Examples

Example 1

The statement `legendre(2,0:0.1:0.2)` returns the matrix

	x = 0	x = 0.1	x = 0.2
m = 0	-0.5000	-0.4850	-0.4400
m = 1	0	-0.2985	-0.5879
m = 2	3.0000	2.9700	2.8800

Example 2

Given,

```
X = rand(2,4,5);
n = 2;
P = legendre(n,X)
```

then

```
size(P)
ans =
     3     2     4     5
```

and

```
P(:,1,2,3)
ans =
   -0.2475
   -1.1225
    2.4950
```

is the same as

```
legendre(n,X(1,2,3))
ans =
   -0.2475
   -1.1225
    2.4950l
```

Algorithms

`legendre` uses a three-term backward recursion relationship in `m`. This recursion is on a version of the Schmidt seminormalized

associated Legendre functions $P_n^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_n(x),$,
which are complex spherical harmonics. These functions are related to the standard Abramowitz and Stegun [1] functions

$P_n^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_n(x),$ by

$$P_n^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_n(x),$$

They are related to the Schmidt form given previously by

$$P_n^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_n(x),$$

References

[1] Abramowitz, M. and I. A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, Ch.8.

[2] Jacobs, J. A., *Geomagnetism*, Academic Press, 1987, Ch.4.

Purpose

Length of vector or largest array dimension

Syntax

```
numberOfElements = length(array)
```

Description

`numberOfElements = length(array)` finds the number of elements along the largest dimension of an array. `array` is an array of any MATLAB data type and any valid dimensions. `numberOfElements` is a whole number of the MATLAB `double` class.

For nonempty arrays, `numberOfElements` is equivalent to `max(size(array))`. For empty arrays, `numberOfElements` is zero.

Examples

Create a 1-by-8 array `X` and use `length` to find the number of elements in the second (largest) dimension:

```
X = [5, 3.4, 72, 28/4, 3.61, 17 94 89];

length(X)
ans =
     8
```

Create a 4-dimensional array `Y` in which the third dimension is the largest. Use `length` to find the number of elements in that dimension:

```
Y = rand(2, 5, 17, 13);

length(Y)
ans =
    17
```

Create a struct array `S` with character and numeric fields of different lengths. Use the `structfun` function to apply `length` to each field of `S`:

```
S = struct('f1', 'Name:', 'f2', 'Charlie', ...
           'f3', 'DOB:', 'f4', 1917)
```

```
S =
    f1: 'Name:'
    f2: 'Charlie'
    f3: 'DOB:'
    f4: 1917

structfun(@(field)length(field), S)
ans =
     5
     7
     4
     1
```

See Also

numel | size | ndims

containers.Map.length

Purpose

Length of `containers.Map` object

Syntax

```
mapLength = length(mapObj)
```

Description

`mapLength = length(mapObj)` returns the number of key-value pairs in `mapObj`.

Input Arguments

mapObj

Object of class `containers.Map`.

Output Arguments

mapLength

Scalar numeric value that indicates the number of key-value pairs in `mapObj`. This number is equivalent to `size(mapObj,1)` and to `mapObj.Count`.

Examples

Determine the Length of a Map

Create a map, and determine the number of key-value pairs:

```
myKeys = {'a','b','c'};
myValues = [1,2,3];
mapObj = containers.Map(myKeys,myValues);

mapLength = length(mapObj)
```

This code returns

```
mapLength =
     3
```

See Also

`containers.Map` | `isKey` | `keys` | `size` | `values`

Purpose

Length of serial port object array

Syntax

```
length(obj)
```

Description

`length(obj)` returns the length of the serial port object, `obj`. It is equivalent to the command `max(size(obj))`.

See Also

`size`

length (tscollection)

Purpose Length of time vector

Syntax

```
length(tsc)
```

Description `length(tsc)` returns an integer that represents the length of the time vector for the `tscollection` object `tsc`.

See Also `tscollection` | `isempty (tscollection)` | `size (tscollection)`

Purpose

Return information on functions in shared library

Syntax

```
libfunctions libname

m = libfunctions(libname)
___ = libfunctions(___,'-full')
```

Description

`libfunctions libname` displays names of functions defined in library, `libname`. If you called `loadlibrary` using the `alias` option, then you must use the alias name for the `libname` argument.

`m = libfunctions(libname)` returns names of functions in cell array `m`.

`___ = libfunctions(___,'-full')` returns function signatures.

Examples

Display Function Signatures

Load the example library, `shrlibsample`, and list the functions.

```
if not(libisloaded('shrlibsample'))
  addpath(fullfile(matlabroot,'extern','examples','shrlib'))
  loadlibrary('shrlibsample')
end
m = libfunctions('shrlibsample','-full')
```

```
m =

    '[double, doublePtr] addDoubleRef(double, doublePtr, double)'
    'double addMixedTypes(int16, int32, double)'
    '[double, c_structPtr] addStructByRef(c_structPtr)'
    'double addStructFields(c_struct)'
    'c_structPtrPtr allocateStruct(c_structPtrPtr)'
    'voidPtr deallocateStruct(voidPtr)'
    'lib.pointer exportedDoubleValue'
    'lib.pointer getListOfStrings'
    'doublePtr multDoubleArray(doublePtr, int32)'
```

```
'[lib.pointer, doublePtr] multDoubleRef(doublePtr)'
'int16Ptr multiplyShort(int16Ptr, int32)'
'doublePtr print2darray(doublePtr, int32)'
'printExportedDoubleValue'
'cstring readEnum(Enum1)'
'[cstring, cstring] stringToUpper(cstring)'
```

Limitations

- Use with libraries that are loaded using the `loadlibrary` function.

See Also

`loadlibrary` | `libfunctionsview` | `calllib`

Purpose

Display shared library function signatures in window

Syntax

```
libfunctionsview libname
```

Description

`libfunctionsview libname` displays information about functions in library, `libname`, in a new window.

Input Arguments

libname - Name of shared library
string

Name of shared library, specified as a string. If you call `loadlibrary` using the `alias` option, then you must use the alias name for the `libname` argument.

Data Types
`char`

Limitations

- Use with libraries that are loaded using the `loadlibrary` function.

Examples

Display Function Signatures for Library libmx

```
if not(libisloaded('libmx'))
  hfile = fullfile(matlabroot,'extern','include','matrix.h');
  loadlibrary('libmx',hfile)
end
libfunctionsview libmx
```

MATLAB creates a new window displaying function signatures.

See Also

`libfunctions` | `calllib`

libisloaded

Purpose

Determine if shared library is loaded

Syntax

```
tf = libisloaded(libname)
```

Description

`tf = libisloaded(libname)` returns logical `1` (`true`) if the shared library, `libname`, is loaded. Otherwise, it returns logical `0` (`false`).

Input Arguments

libname - Name of shared library

string

Name of shared library, specified as a string. If you call `loadlibrary` using the `alias` option, then you must use the alias name for the `libname` argument.

Data Types

`char`

Examples

Load Functions in Library

Load example library, `shrlibsample`, if it is not already loaded.

```
if ~libisloaded('shrlibsample')
  loadlibrary('shrlibsample')
end
```

See Also

`loadlibrary` | `unloadlibrary`

Purpose

Pointer object for use with shared library

Syntax

```
p = libpointer
p = libpointer(DataType)
p = libpointer(DataType,Value)
```

Description

`p = libpointer` creates `NULL` pointer `p`.

`p = libpointer(DataType)` creates an empty pointer of specified `DataType`.

`p = libpointer(DataType,Value)` creates a pointer initialized to `Value`.

Limitations

- Use with libraries that are loaded using the `loadlibrary` function.

Input Arguments

DataType - Type of pointer

string

Type of pointer, specified as a string, of any MATLAB numeric type, structure defined in the library, or enumeration defined in the library. For a list of valid MATLAB numeric types, see "Equivalent MATLAB Type" in the tables "MATLAB Primitive Types" and "MATLAB Extended Types".

Example: `p = libpointer('int16Ptr')`

Data Types

`char`

Value - Value for pointer object

Any valid value

Value, specified as any valid value for given type.

Example:
```
v = [4 6 8; 7 5 3];
p = libpointer('int16Ptr',v);
```

Tips

- This is an advanced feature for experienced C programmers. MATLAB automatically converts data passed to and from external library functions to the data type expected by the external function. Use the `libpointer` function when you want to manually convert argument data, using the `setdatatype` method.

See Also

`calllib` | `libstruct`

Related Examples

- "Multilevel Pointers"

Concepts

- "Reading a libpointer Object"
- "Working with Pointer Arguments"

Purpose

Pointer object compatible with C pointer

Description

When a function in an external library passes arguments by reference (indicated in the function signature with names ending in `Ptr` and `PtrPtr`), MATLAB automatically converts. In the following cases,

- Modify data in an input argument.
- Not copy data passed to a function.
- Control the lifetime of a `libpointer` object.

Construction

To create a `lib.pointer` object, use the MATLAB `libpointer` function.

Properties

Value

DataType

Methods

disp

isnull

plus

reshape

setdatatype

Definitions

A `libpointer` is an instance of the lib.pointer class.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Alternatives

You do not need to create a `libpointer` object for every function with pointer arguments. MATLAB automatically converts MATLAB variables for these argument types.

See Also

`libpointer`

Concepts

- "Working with Pointer Arguments"

Purpose

Syntax

```
disp(h)
```

Description

```
disp(h)
```

Tips

-

Input Arguments

h

handle

Examples

lib.pointer.isNull

Purpose

Syntax

```
tf = isNull(h)
```

Description

```
tf = isNull(h)
```

Tips

-

Input Arguments

h

handle

Examples

Purpose

Syntax

```
h = plus(h,offset)
```

Description

```
h = plus(h,offset)
```

Tips

-

Input Arguments

h

handle

offset

uint64

Output Arguments

h

handle

Examples

lib.pointer.reshape

Purpose

Syntax

`reshape(h,xdim,ydim)`

Description

`reshape(h,xdim,ydim)`

Tips

•

Input Arguments

h
handle

xdim
double

ydim
double

Examples

Purpose

Syntax

`setdatatype(h,type,size)`

Description

`setdatatype(h,type,size)`

Tips

-

Input Arguments

h
handle

type
string

size
double

Examples

libstruct

Purpose

Convert MATLAB structure to C-style structure for use with shared library

Syntax

```
S = libstruct(structtype)
S = libstruct(structtype,mlstruct)
```

Description

`S = libstruct(structtype)` creates `NULL` pointer to MATLAB `libstruct` object `S`.

`S = libstruct(structtype,mlstruct)` creates pointer initialized to `mlstruct`.

Limitations

- Use with libraries that are loaded using the `loadlibrary` function.
- You can only use the `libstruct` function on scalar structures.
- When converting a MATLAB structure to a `libstruct` object, the structure must adhere to the requirements listed in "Structure Argument Requirements".

Input Arguments

structtype - C structure
structure

C structure defined in shared library.

mlstruct - MATLAB structure
structure

MATLAB structure used to initialize the fields in `S`.

Data Types
`struct`

Output Arguments

S - Pointer
MATLAB `libstruct` object

Pointer, returned as MATLAB `libstruct` object.

Examples

Call Function with c_struct Input Argument

Call the `addStructFields` function by creating a variable of type `c_struct`.

Load the `shrlibsample` library, which contains the `c_struct` type.

```
if ~libisloaded('shrlibsample')
   addpath(fullfile(matlabroot,'extern','examples','shrlib'))
   loadlibrary('shrlibsample')
end
```

Display function signatures for `shrlibsample` and search the list for the `addStructFields` entry.

```
libfunctionsview shrlibsample
```

```
double addStructFields(c_struct)
```

The input argument is a pointer to a `c_struct` data type.

Create a MATLAB structure, `sm`.

```
sm.p1 = 476;   sm.p2 = -299;   sm.p3 = 1000;
```

Construct a `libstruct` object `sc` from the `c_struct` type.

```
sc = libstruct('c_struct',sm)
```

The fields of `sc` contain the values of the MATLAB structure, `sm`.

Call the `addStructFields` function.

```
calllib('shrlibsample','addStructFields',sc)
```

```
ans =
    1177
```

To clean up, first clear the `libstruct` object, and then unload the library.

```
clear sc
unloadlibrary shrlibsample
```

Tips

- If a function in the shared library has a structure argument, use `libstruct` to create the argument. The `libstruct` function creates a C-style structure that you pass to functions in the library. You handle this structure in MATLAB as you would a true MATLAB structure.

See Also

`loadlibrary` | `libfunctionsview`

Concepts

- "Working with Structure Arguments"

Purpose

Return license number or perform licensing task

Syntax

```
license
license('inuse')
S = license('inuse')
S = license('inuse', feature)
license('test',feature)
license('test',feature,toggle)
[TF errmsg] = license('checkout',feature)
```

Description

`license` returns the license number for this MATLAB product. The return value is always a string but is not guaranteed to be a number. The following table lists text strings that `license` can return.

String	Description
`'demo'`	MATLAB is a demonstration version
`'student'`	MATLAB is the student version
`'unknown'`	License number cannot be determined

`license('inuse')` returns a list of licenses checked out in the current MATLAB session. In the list, products are listed alphabetically by their license feature names, i.e., the text string used to identify products in the `INCREMENT` lines in a License File (`license.dat`). Note that the feature names returned in the list contain only lower-case characters.

`S = license('inuse')` returns an array of structures, where each structure represents a checked-out license. The structures contains two fields: feature and user. The feature field contains the license feature name. The user field contains the username of the person who has the license checked out.

`S = license('inuse', feature)` checks if the product specified by the text string `feature` is checked out in the current MATLAB session. If the product is checked out, the `license` function returns the product name and the username of the person who has it checked out in the

structure `S`. If the product is not currently checked out, the fields in the structure are empty.

The `feature` string must be a license feature name, spelled exactly as it appears in the `INCREMENT` lines in a License File. For example, the string `'Identification_Toolbox'` is the feature name for the System Identification Toolbox™. The `feature` string is not case-sensitive and must not exceed 27 characters.

`license('test',feature)` tests if a license exists for the product specified by the text string `feature`. The `license` command returns `1` if the license exists and `0` if the license does not exist. The `feature` string identifies a product, as described in the previous syntax.

Note Testing for a license only confirms that the license exists. It does not confirm that the license can be checked out. For example, license will return 1 if a license exists, even if the license has expired or if a system administrator has excluded you from using the product in an options file. The existence of a license does not indicate that the product is installed.

`license('test',feature,toggle)` enables or disables testing of the product specified by the text string `feature`, depending on the value of `toggle`. The parameter `toggle` can have either of two values:

`'enable'`	The syntax `license('test',feature)` returns `1` if the product license exists and 0 if the product license does not exist.
`'disable'`	The syntax `license('test',feature)` always returns `0` (product license does not exist) for the specified product.

Note Disabling a test for a particular product can impact other tests for the existence of the license, not just tests performed using the `license` command.

`[TF errmsg] = license('checkout',feature)` checks out a license for the product specified by the text string `feature`, returning `1` if it could check out a license or `0` if it could not check out a license. If you specify the optional second output argument, `errmsg`, the `license` function returns the text of any error message encountered, or an empty string if the checkout succeeded.

Examples

Get the license number for this MATLAB.

```
license
```

Get a list of licenses currently being used. Note that the products appear in alphabetical order by their license feature name in the list returned.

```
license('inuse')

image_toolbox
map_toolbox
matlab
```

Get a list of licenses in use with information about who is using the license.

```
S = license('inuse');

S(1)

ans =

    feature: 'image_toolbox'
       user: 'juser'
```

Determine if the license for MATLAB is currently in use.

```
S = license('inuse','MATLAB')

S =

    feature: 'matlab'
       user: 'jsmith'
```

Determine if a license exists for the Mapping Toolbox™.

```
license('test','map_toolbox')

ans =

    1
```

Check out a license for the Control System Toolbox™.

```
license('checkout','control_toolbox')

ans =

    1
```

Determine if the license for the Control System Toolbox is checked out.

```
license('inuse')

control_toolbox
image_toolbox
map_toolbox
matlab
```

See Also

`isstudent`

Purpose

Create light object

Syntax

```
light('PropertyName',propertyvalue,...)
handle = light(...)
```

Properties

For a list of properties, see Light Properties.

Description

`light` creates a light object in the current axes. Lights affect only patch and surface objects.

`light('PropertyName',propertyvalue,...)` creates a light object using the specified values for the named properties. For a description of the properties, see Light Properties. The MATLAB software parents the light to the current axes unless you specify another axes with the `Parent` property.

`handle = light(...)` returns the handle of the light object created.

Tips

You cannot see a light object *per se*, but you can see the effects of the light source on patch and surface objects. You can also specify an axes-wide ambient light color that illuminates these objects. However, ambient light is visible only when at least one light object is present and visible in the axes.

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see `set` and `get` for examples of how to specify these data types).

See also the patch and surface `AmbientStrength`, `DiffuseStrength`, `SpecularStrength`, `SpecularExponent`, `SpecularColorReflectance`, and `VertexNormals` properties. Also see the `lighting` and `material` commands.

Examples

Light the `peaks` surface plot with a light source located at infinity and oriented along the direction defined by the vector `[1 0 0]`, that is, along the *x*-axis.

```
h = surf(peaks);
```

```
set(h,'FaceLighting','phong','FaceColor','interp',...
      'AmbientStrength',0.5)
light('Position',[1 0 0],'Style','infinite');
```

Setting Default Properties

You can set default light properties on the axes, figure, and `root object` levels:

```
set(0,'DefaultLightProperty',PropertyValue...)
set(gcf,'DefaultLightProperty',PropertyValue...)
set(gca,'DefaultLightProperty',PropertyValue...)
```

where *`Property`* is the name of the light property and `PropertyValue` is the value you are specifying. Use `set` and `get` to access light properties.

Tutorials

For more information about lighting, see "Lighting Overview".

See Also

`lighting` | `material` | `patch` | `surface` | Light Properties

Purpose

Light properties

Creating Light Objects

Use `light` to create light objects.

Modifying Properties

You can set and query graphics object properties in two ways:

- "The Property Editor" is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see "Setting Default Property Values".

See "Core Graphics Objects" for general information about this type of object.

Light Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`BeingDeleted`

`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`
function handle

This property is not used on lights.

`Children`
handles

The empty matrix; light objects have no children.

`Clipping`
`on | off`

Clipping mode. This property has no effect on light objects.

`Color`
`ColorSpec`

Light color. Defines the color of the light emanating from the light object. Use a three-element RGB vector or one of the MATLAB predefined names. Default value is `[1 1 1]` (white). See the `ColorSpec` reference page for more information on specifying color.

`CreateFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback function executed during object creation. Executes when MATLAB creates a light object. You must define this property as a default value for lights or in a call to the `light` function to create a new light object. For example, the following statement:

```
set(0,'DefaultLightCreateFcn',@light_create)
```

defines a default value for the line `CreateFcn` property on the root level that sets the current figure colormap to `gray` and uses a reddish light color whenever you create a light object.

```
function light_create(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
 set(src,'Color',[.9 .2 .2])
 set(gcbf,'Colormap',gray)
end
```

MATLAB executes this function after setting all light properties. Setting this property on an existing light object has no effect. The function must define at least two input arguments (handle of light object created and an event structure, which is empty for this property).

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DeleteFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Delete light callback function. Executes when you delete the light object (for example, when you issue a `delete` command or clear the axes `cla` or figure `clf`).

For example, the following function displays object property data before the object is deleted.

```
function delete_fcn(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   obj_tp = get(src,'Type');
   disp([obj_tp, ' object deleted'])
   disp('Its user data is:')
   disp(get(src,'UserData'))
end
```

MATLAB executes the function before deleting the object's properties so these values are available to the callback function. The function must define at least two input arguments (handle of object being deleted and an event structure, which is empty for this property).

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

- `on` — Handles are always visible.
- `callback` — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

`HitTest`
`{on} | off`

This property is not used by light objects.

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the `Interruptible` property affects only the callbacks for the`ButtonDownFcn` property. A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both the callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see "Control Callback Execution and Interruption".

`Parent`

handle of parent axes

Parent of light object. This property contains the handle of the light object's parent. The parent of a light object is the `axes` object that contains it.

Note that light objects cannot be parented to hggroup or hgtransform objects.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Position`

`[x,y,z]` in axes data units

Location of light object. Vector defining the location of the light object. The vector is defined from the origin to the specified *x*-,

y-, and *z*-coordinates. The placement of the light depends on the setting of the `Style` property:

- If the `Style` property is `local`, `Position` specifies the actual location of the light (which is then a point source that radiates from the location in all directions).
- If the `Style` property is `infinite`, `Position` specifies the direction from which the light shines in parallel rays.

`Selected`
`on | off`

This property is not used by light objects.

`SelectionHighlight`
`{on} | off`

This property is not used by light objects.

`Style`
`{infinite} | local`

Parallel or divergent light source.

- `infinite` — MATLAB places the light object at infinity (light rays are parallel).
- `local` — MATLAB places the light object at the location specified by the `Position` property (light rays diverge in all directions).

`Tag`
string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

`Type`

string (read-only)

Type of graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For light objects, `Type` is always '`light`'.

`UIContextMenu`

handle of uicontextmenu object

This property is not used by light objects.

`UserData`

matrix

User-specified data. Data you want to associate with the light object. The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`

`{on} | off`

Light visibility. While light objects themselves are not visible, you can see the light on patch and surface objects. When you set `Visible` to `off`, the light emanating from the source is not visible. There must be at least one light object in the axes whose `Visible` property is `on` for any lighting features to be enabled (including the axes `AmbientLightColor` and patch and surface `AmbientStrength`).

See Also

`light`

lightangle

Purpose

Create or position light object in spherical coordinates

Syntax

```
lightangle(az,el)
light_handle = lightangle(az,el)
lightangle(light_handle,az,el)
[az,el] = lightangle(light_handle)
```

Description

`lightangle(az,el)` creates a light at the position specified by azimuth and elevation. `az` is the azimuthal (horizontal) rotation and `el` is the vertical elevation (both in degrees). The interpretation of azimuth and elevation is the same as that of the `view` command.

`light_handle = lightangle(az,el)` creates a light and returns the handle of the light in `light_handle`.

`lightangle(light_handle,az,el)` sets the position of the light specified by `light_handle`.

`[az,el] = lightangle(light_handle)` returns the azimuth and elevation of the light specified by `light_handle`.

Tips

By default, when a light is created, its style is `infinite`. If the light handle passed in to `lightangle` refers to a local light, the distance between the light and the camera target is preserved as the position is changed.

Examples

```
surf(peaks)
axis vis3d
h = light;
for az = -50:10:50
 lightangle(h,az,30)
 drawnow
end
```

See Also

`light` | `camlight` | `view`

How To

- "Lighting Overview"

Purpose

Specify lighting algorithm

Syntax

```
lighting flat
lighting gouraud
lighting phong
lighting none
```

Description

`lighting` selects the algorithm used to calculate the effects of `light` objects on all `surface` and `patch` objects in the current axes. In order for the `lighting` command to have any effects, however, you must create a lighting object by using the `light` function.

`lighting flat` produces uniform lighting across each of the faces of the object. Select this method to view faceted objects.

`lighting gouraud` calculates the vertex normals and interpolates linearly across the faces. Select this method to view curved surfaces.

`lighting phong` interpolates the vertex normals across each face and calculates the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but it takes longer to render.

`lighting none` turns off lighting.

Tips

The `surf`, `mesh`, `pcolor`, `fill`, `fill3`, `surface`, and `patch` functions create graphics objects that are affected by light sources. The `lighting` command sets the `FaceLighting` and `EdgeLighting` properties of surfaces and patches appropriately for the graphics object.

See Also

`fill` | `fill3` | `light` | `material` | `mesh` | `patch` | `pcolor` | `shading` | `surface`

How To

- "Lighting Overview"

Purpose

Convert linear audio signal to mu-law

Syntax

```
mu = lin2mu(y)
```

Description

`mu = lin2mu(y)` converts linear audio signal amplitudes in the range `-1    Y    1` to mu-law encoded “flints” in the range `0    u    255`.

See Also

`auwrite` | `mu2lin`

Purpose

Create line object

Syntax

```
line
line(X,Y)
line(X,Y,Z)
line(X,Y,Z,'PropertyName',propertyvalue,...)
line('XData',x,'YData',y,'ZData',z,...)
h = line(...)
```

Properties

For a list of properties, see Line Properties.

Description

`line` creates a line object in the current axes with default values `x = [0 1]` and `y = [0 1]`. You can specify the color, width, line style, and marker type, as well as other characteristics.

The `line` function has two forms:

- Automatic color and line style cycling. When you specify multiple line coordinate data as a column array using the informal syntax (i.e., the first three arguments are interpreted as the coordinates),

  ```
  line(X,Y,Z)
  ```

  MATLAB cycles through the axes `ColorOrder` and `LineStyleOrder` property values the way the `plot` function does. However, unlike `plot`, `line` does not call the `newplot` function.

- Purely low-level behavior. When you call `line` with only property name/property value pairs,

  ```
  line('XData',x,'YData',y,'ZData',z)
  ```

  MATLAB draws a line object in the current axes using the default line color (see the `colordef` function for information on color defaults). Note that you cannot specify matrix coordinate data with the low-level form of the `line` function.

`line(X,Y)` adds the line defined in vectors `X` and `Y` to the current axes. If `X` and `Y` are matrices of the same size, `line` draws one line per column.

`line(X,Y,Z)` creates lines in three-dimensional coordinates.

`line(X,Y,Z,'PropertyName',propertyvalue,...)` creates a line using the values for the property name/property value pairs specified and default values for all other properties. For a description of the properties, see Line Properties.

See the `LineStyle` and `Marker` properties for a list of supported values.

`line('XData',x,'YData',y,'ZData',z,...)` creates a line in the current axes using the property values defined as arguments. This is the low-level form of the `line` function, which does not accept matrix coordinate data as the other informal forms described above.

`h = line(...)` returns a column vector of handles corresponding to each line object the function creates.

Tips

In its informal form, the `line` function interprets the first three arguments (two for 2-D) as the `X`, `Y`, and `Z` coordinate data, allowing you to omit the property names. You must specify all other properties as name/value pairs. For example,

```
line(X,Y,Z,'Color','r','LineWidth',4)
```

The low-level form of the `line` function can have arguments that are only property name/property value pairs. For example,

```
line('XData',x,'YData',y,'ZData',z,'Color','r','LineWidth',4)
```

Line properties control various aspects of the line object and are described in Line Properties. You can also set and query property values after creating the line using `set` and `get`.

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the `set` and `get` reference pages for examples of how to specify these data types).

Unlike high-level functions such as `plot`, `line` does not respect the settings of the figure and axes `NextPlot` properties. It simply adds line objects to the current axes. However, axes properties that are under

automatic control, such as the axis limits, can change to accommodate the line within the current axes.

Connecting the dots

The coordinate data is interpreted as vectors of corresponding x, y, and z values:

```
X = [x(1) x(2) x(3)...x(n)]
Y = [y(1) y(2) y(3)...y(n)]
Z = [z(1) z(2) z(3)...z(n)]
```

where a point is determined by the corresponding vector elements:

```
p1(x(i),y(i),z(i))
```

For example, to draw a line from the point located at `x = .3` and `y = .4` and `z = 1` to the point located at `x = .7` and `y = .9` and `z = 1`, use the following data:

```
axis([0 1 0 1])
line([.3 .7],[.4 .9],[1 1],'Marker','.','LineStyle','-')
```

Examples

This example uses the `line` function to add a shadow to plotted data. First, plot some data and save the line's handle:

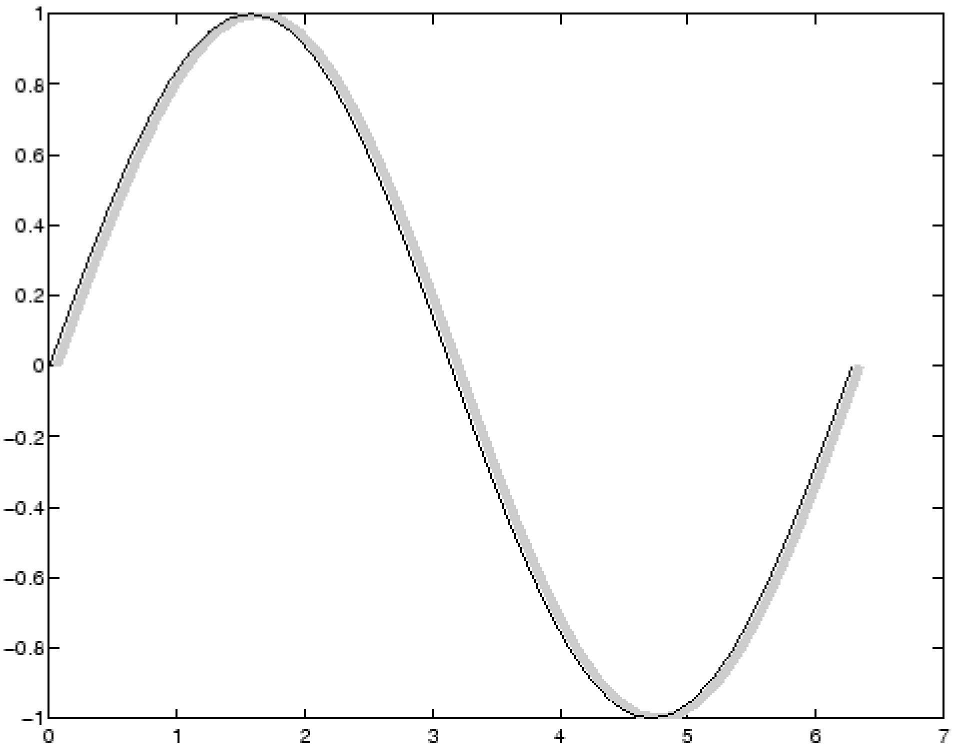
```
t = 0:pi/20:2*pi;
hline1 = plot(t,sin(t),'k');
```

Next, add a shadow by offsetting the *x*-coordinates. Make the shadow line light gray and wider than the default `LineWidth`:

```
hline2 = line(t+.06,sin(t),'LineWidth',4,'Color',[.8 .8 .8]);
```

Finally, pull the first line to the front:

```
set(gca,'Children',[hline1 hline2])
```



Drawing Lines Interactively

You can use the `ginput` function to select points from a figure. For example:

```
[x,y] = ginput(5);
line(x,y)
```

Drawing with mouse motion

You can use the axes `CurrentPoint` property and the figure `WindowButtonDownFcn` and `WindowButtonMotionFcn` properties to select a point with a mouse click and draw a line to another point by dragging the mouse, like a simple drawing program. The following

example illustrates a few useful techniques for doing this type of interactive drawing.

Click to view in editor — This example enables you to click and drag the cursor to draw lines.

Click to run example — Click the left mouse button in the axes and move the cursor, left-click to define the line end point, right-click to end drawing mode.

Input Argument Dimensions — Informal Form

This statement reuses the one-column matrix specified for `ZData` to produce two lines, each having four points.

```
line(rand(4,2),rand(4,2),rand(4,1))
```

If all the data has the same number of columns and one row each, MATLAB transposes the matrices to produce data for plotting. For example,

```
line(rand(1,4),rand(1,4),rand(1,4))
```

is changed to

```
line(rand(4,1),rand(4,1),rand(4,1))
```

This also applies to the case when just one or two matrices have one row. For example, the statement

```
line(rand(2,4),rand(2,4),rand(1,4))
```

is equivalent to

```
line(rand(4,2),rand(4,2),rand(4,1))
```

Setting Default Properties

You can set default line properties on the axes, figure, and `root object` levels:

```
set(0,'DefaultLinePropertyName',PropertyValue,...)
set(gcf,'DefaultLinePropertyName',PropertyValue,...)
```

```
set(gca,'DefaultLinePropertyName',PropertyValue,...)
```

Where *PropertyName* is the name of the line property and `PropertyValue` is the value you are specifying. Use `set` and `get` to access line properties.

See Also

`annotation` | `axes` | `newplot` | `plot` | `plot3` | Line Properties

Purpose

Line properties

Creating Line Objects

Use `line` to create line objects.

Modifying Properties

You can set and query graphics object properties in two ways:

- "The Property Editor" is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see "Setting Default Property Values".

See Core Graphics Objects for general information about this type of object.

Line Property Descriptions

`Annotation`

`hg.Annotation` object (read-only)

Handle of `Annotation` object. The `Annotation` property enables you to specify whether this line object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the line object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Represent this line object in a legend (default)
`off`	Do not include this line object in a legend
`children`	Same as `on` because line objects do not have children

Setting the IconDisplayStyle property

Set the `IconDisplayStyle` of a graphics object with handle `hobj` to `off`:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Using the IconDisplayStyle property

See “Controlling Legends” for more information and examples.

`BeingDeleted`

`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object’s delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object’s delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object’s `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Button press callback function. Executes whenever you press a mouse button while the pointer is over the line object. The default is an empty array.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

Set this property to a function handle that references the callback. The function must define at least two input arguments (handle

of line associated with the button down event and an event structure, which is empty for this property).

The following example shows how to access the callback object's handle as well as the handle of the figure that contains the object from the callback function.

```
function button_down(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   sel_typ = get(gcbf,'SelectionType')
   switch sel_typ
      case 'normal'
         disp('User clicked left-mouse button')
         set(src,'Selected','on')
      case 'extend'
         disp('User did a shift-click')
         set(src,'Selected','on')
      case 'alt'
         disp('User did a control-click')
         set(src,'Selected','on')
         set(src,'SelectionHighlight','off')
   end
end
```

Suppose `h` is the handle of a line object and the `button_down` function is on your MATLAB path. The following statement assigns the `button_down` function to the `ButtonDownFcn` property:

```
set(h,'ButtonDownFcn',@button_down)
```

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`Children`
vector of handles

The empty matrix; line objects have no children.

`Clipping`
`{on} | off`

Clipping mode. MATLAB clips lines to the axes plot box by default. If you set `Clipping` to `off`, lines are displayed outside the axes plot box. This occurs if you create a line, set `hold` to `on`, freeze axis scaling (set `axis` to `manual`), and then create a longer line.

`Color`
`ColorSpec`

Line color. A three-element RGB vector, or one of the MATLAB predefined names, specifying the line color. See the `ColorSpec` reference page for more information on specifying color.

`CreateFcn`
function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback function executed during object creation. Executes when MATLAB creates a line object. The default is an empty array.

You must specify the callback during the creation of the line or in a call to the `line` function to create a new line object.

For example, the statement:

```
set(0,'DefaultLineCreateFcn',@line_create)
```

defines a default value for the line `CreateFcn` property on the root level that sets the axes `LineStyleOrder` whenever you create a line object. The following callback function must be on your MATLAB path when you execute the above statement.

```
function line_create(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
 axh = get(src,'Parent');
 set(axh,'LineStyleOrder','-.|--')
```

```
end
```

MATLAB executes this function after setting all line properties. Setting this property on an existing line object has no effect.

The function must define at least two input arguments (handle of line object created and an event structure, which is empty for this property).

The handle of the object whose `CreateFcn` is being executed is MATLAB as the first argument to the callback function and is also accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DeleteFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Delete line callback function. Executes when you delete the line object. This happens when you call the `delete` command on the object, its parent axes, or the figure containing it. The default is an empty array.

For example, the following function displays object property data before the object is deleted.

```
function delete_fcn(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   obj_tp = get(src,'Type');
   disp([obj_tp, ' object deleted'])
   disp('Its user data is:')
   disp(get(src,'UserData'))
end
```

MATLAB executes the function before deleting the object's properties so these values are available to the callback function. The function must define at least two input arguments (handle of line object being deleted and an event structure, which is empty for this property).

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DisplayName`
string

String used by legend. The `legend` function uses the `DisplayName` property to label the line object in the legend. The default is an empty string.

- If you specify string arguments with the `legend` function, MATLAB set `DisplayName` to the corresponding string and uses that string for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, MATLAB set `DisplayName` to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add a legend programmatically that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more information and examples.

The following code shows how to use the `DisplayName` property from the command line or in a file.

```
t = 0:.1:2*pi;
a(:,1)=sin(t); a(:,2)=cos(t);
h = plot(a);
set(h,{'DisplayName'},{'Sine','Cosine'}')
legend show
```

`EraseMode`

`{normal} | none | xor | background`

Erase mode. Controls the technique MATLAB uses to draw and erase line objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the line when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it, because MATLAB stores no information about its former location.
- `xor` — Draw and erase the line by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath the line. However, the line's color depends on the color of whatever is beneath it on the display.
- `background` — Erase the line by drawing it in the axes background `Color`, or the figure background `Color` if the axes

`Color` is `none`. This damages objects that are behind the erased line, but lines are always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors (for example, performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`HandleVisibility`
`{on} | callback | off`

Control access to object handle. Determine when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

- `on` — Handles are always visible.
- `callback` — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

For more information, see "HandleVisibility Property", "Functions Affected by Handle Visibility", and "Properties Affected by Handle Visibility".

`HitTest`

`{on} | off`

Selectable by mouse click. Determines if the line can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the line. If `HitTest` is `off`, clicking the line selects the object below it (which might be the axes containing it).

`Interruptible`

`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the `Interruptible` property affects only the callbacks for the`ButtonDownFcn` property. A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both the callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see "Control Callback Execution and Interruption".

`LineStyle`

`{-} | -- | : | -. | none`

Line style.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line
`'none'`	No line

Use `LineStyle` `none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`

scalar

Width of the line . Specify this value in points. The default value is `0.5` points.

`Marker`
character (see table)

Marker symbol. Specifies marks displayed at data points. You can set values for the `Marker` property independently from the `LineStyle` property. For a list of supported marker symbols, see the following table.

Marker Specifiers Table

Specifier	Marker Type
'+'	Plus sign
'o'	Circle
'*'	Asterisk
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'''	Six-pointed star (hexagram)
'none'	No marker (default)

`MarkerEdgeColor`
`ColorSpec | none | {auto}`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `ColorSpec` — User-defined color.
- `none` — Specifies no color, which makes nonfilled markers invisible.
- `auto` — Use same color as the line `Color` property.

`MarkerFaceColor`
`ColorSpec | {none} | auto`

Fill color for closed-shape markers. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `ColorSpec` — User-defined color.
- `none` — Makes the interior of the marker transparent, allowing the background to show through.
- `auto` — Sets the fill color to the axes `Color` property. If the axes `Color` property is `none`, sets the fill color to the figure `Color`.

`MarkerSize`
scalar

Marker size. Size of the marker in points. The default value is `6`.

Note MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`Parent`
handle of axes, hggroup, or hgtransform

Parent of line object. Handle of the line object's parent. The parent of a line object is the axes that contains it. You can reparent line objects to other axes, hggroup, or hgtransform objects.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | off`

Selection status When this property is `on`, MATLAB displays selection handles if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`
`{on} | off`

Object highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing handles at each vertex. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`Tag`
string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use this property when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callbacks. You can define `Tag` as any string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

`Type`
string (read-only)

Class of graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For line objects, `Type` is always '`line`'.

`UIContextMenu`
handle of uicontextmenu object

Associate context menu with line. Assign this property the handle of a uicontextmenu object created in the same figure as the line. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the line.

`UserData`
matrix

User-specified data. Data you want to associate with the line object. The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

Note Setting the `UserData` property more than once erases previous entries.

`Visible`
`{on} | off`

Line visibility.

- `on` — All lines are visible.
- `off` — Line is not visible, but still exists, and you can `get` and `set` its properties.

`XData`
vector

x-coordinates of line. Vector of *x*-coordinates defining the line. The default is `[0 1]`.

`YData` and `ZData`, if set, must be the same length and have the same number of rows.
Create a line using the `line` function, then change its `XData` property:

```
mylin = line(rand(1,10),rand(1,10));
set(mylin,'XData',rand(1,15));
% A warning appears, since you did not set the YData property.
set(mylin,'XData',rand(1,15),'YData',rand(1,15));
% No warning appears and the plot draws as expected.
```

`YData`

vector

y-coordinates of line. Vector of *y*-coordinates defining the line. The default is `[0 1]`. `XData` and `ZData`, if set, must be the same length and have the same number of rows.
Create a line using the `line` function, then change its `YData` property:

```
mylin = line(rand(1,10),rand(1,10));
set(mylin,'YData',rand(1,15));
% A warning appears, since you did not set the YData property.
set(mylin,'YData',rand(1,15),'XData',rand(1,15));
% No warning appears and the plot draws as expected.
```

`ZData`

vector

z-coordinates of line. Vector of *z*-coordinates defining the line. The default is `[]` (empty). `XData` and `YData` must have the same number of rows.
Create a line using the `line` function, then change its `ZData` property:

```
mylin = line(rand(1,10),rand(1,10),rand(1,10));
view(3)
set(mylin,'ZData',rand(1,15));
% A warning appears, since you did not set the YData property.
set(mylin,'XData',rand(1,15),'YData',rand(1,15),'ZData',rand(1,15));
% No warning appears and the plot draws as expected.
```

See Also

line

Purpose

Description of lineseries properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands or with the property editor (`propertyeditor`).

See "Plot Objects" for more information on lineseries objects.

Note that you cannot define default properties for lineseries objects.

Lineseries Property Descriptions

This section lists property names along with the type of values each accepts. Curly braces { } enclose default values.

`Annotation`

`hg.Annotation` object (read-only)

Control the display of lineseries objects in legends. Specifies whether this lineseries object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the lineseries object is displayed in a figure legend.

IconDisplayStyle Value	Purpose
`on`	Include the lineseries object in a legend as one entry, but not its children objects
`off`	Do not include the lineseries or its children in a legend (default)
`children`	Include only the children of the lineseries as separate entries in the legend

Setting the IconDisplayStyle Property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the IconDisplayStyle Property

See "Controlling Legends" for more information and examples.

`BeingDeleted`

`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the

running callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`
string | function handle

Button press callback function. Executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be:

- A string that is a valid MATLAB expression
- The name of a MATLAB file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`Children`
vector of handles

The empty matrix; line objects have no children.

`Clipping`

`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs appear outside the axes plot box. This occurs if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`Color`

ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object's color. The default value is `[0 0 0]` (`black`).

See the `ColorSpec` reference page for more information on specifying color. See "Adding Arrows and Lines to Graphs".

`CreateFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback function executed during object creation. Defines a callback that executes when MATLAB creates an object. The default is an empty array.

You must specify the callback during the creation of the object. For example:

```
graphicfcn(y,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function and `graphicfcn` is the plotting function which creates this object.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DeleteFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback executed during object deletion. Executes when this object is deleted (for example, this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values. The default is an empty array.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`

string

String used by legend. The `legend` function uses the `DisplayName` property to label the lineseries object in the legend. The default is an empty string.

- If you specify string arguments with the `legend` function, MATLAB set `DisplayName` to the corresponding string and uses that string for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, MATLAB set `DisplayName` to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add a legend programmatically that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See “Controlling Legends” for more information and examples.

`EraseMode`

`{normal} | none | xor | background`

Erase mode. Controls the technique MATLAB uses to draw and erase objects. Use alternative erase modes for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the object when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it because MATLAB stores no information about its former location.

- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object's color depends on the color of whatever is beneath it on the display.
- `background` — Erase the object by redrawing it in the axes background `Color`, or the figure background `Color` if the axes `Color` property is `none`. This damages objects that are behind the erased object, but properly colors the erased object.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors (for example, performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible.

- `callback` — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties. See also `findall`.

Handle Validity

Hidden handles are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`
`{on} | off`

Selectable by mouse click. Determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the line. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the `Interruptible` property affects only the callbacks for the`ButtonDownFcn` property. A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both the callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see “Control Callback Execution and Interruption”.

`LineStyle`

`{-} | -- | :  | -.  | none`

Line style of lineseries object.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line
'none'	No line

Use `LineStyle` `none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`
size in points

Width of linear objects and edges of filled areas. Specify in points. 1 point = $^1/_{72}$ inch. The default is `0.5` points.

`Marker`
character (see table)

Marker symbol. Specifies marks that display at data points. You can set values for the `Marker` property independently from the `LineStyle` property. For a list of supported marker symbols, see the following table.

Marker Specifiers Table

Specifier	Marker Type
'+'	Plus sign
'o'	Circle
'*'	Asterisk

Specifier	Marker Type
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'''	Six-pointed star (hexagram)
'none'	No marker (default)

`MarkerEdgeColor`

`ColorSpec | none | {auto}`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `ColorSpec` — User-defined color.
- `none` — Specifies no color, which makes nonfilled markers invisible.
- `auto` — Uses same color as the `Color` property.

`MarkerFaceColor`

`ColorSpec | {none} | auto`

Fill color for closed-shape markers. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `ColorSpec` — User-defined color.
- `none` — Makes the interior of the marker transparent, allowing the background to show through.
- `auto` — Sets the fill color to the axes `Color` property. If the axes `Color` property is `none`, sets the fill color to the figure `Color`.

`MarkerSize`
scalar

Marker size. Size of the marker in points. The default value is `6`.

Note MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`Parent`
handle of parent axes, hggroup, or hgtransform

Parent of object. Handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Object selection state. When you set this property to `on`, MATLAB displays selection handles at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`

`{on} | off`

Object highlighted when selected.

- `on` — MATLAB indicates the selected state by drawing four edge handles and four corner handles.
- `off` — MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`Tag`

string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

For example, create an areaseries object and set the `Tag` property.

```
t = area(Y,'Tag','area1')
```

When you want to access objects of a given type, use `findobj` to find the object's handle. The following statement changes the `FaceColor` property of the object whose `Tag` is `area1`.

```
set(findobj('Tag','area1'),'FaceColor','red')
```

`Type`

string (read-only)

Class of graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For lineseries objects, `Type` is always the string `line`.

`UIContextMenu`

handle of uicontextmenu object

Associate context menu with object. Handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object. The default value is an empty array.

`UserData`
array

User-specified data. Data you want to associate with this object (including cell arrays and structures). The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`
`{on} | off`

Visibility of object and its children.

- `on` — Object and all children of the object are visible unless the child object's `Visible` property is `off`.
- `off` — Object not displayed. However, the object still exists and you can set and query its properties.

`XData`
vector | matrix

x-axis values for graph. The *x*-axis values for graphs are specified by the `X` input argument. If `XData` is a vector, `length(XData)` must equal `length(YData)` and must be monotonic. If `XData` is a matrix, `size(XData)` must equal `size(YData)` and each column must be monotonic.

You can use `XData` to define meaningful coordinates for an underlying surface whose topography is being mapped. See "Changing the Offset of a Contour" for more information.

`XDataMode`
`{auto} | manual`

Use automatic or user-specified x-axis values. If you specify `XData` (by setting the `XData` property or specifying the `X` input argument), MATLAB sets this property to `manual` and uses the specified values to label the *x*-axis.

If you set `XDataMode` to `auto` after specifying `XData`, MATLAB resets the *x*-axis ticks to `1:size(YData,1)` or to the column indices of the `ZData`, overwriting any previous values for `XData`.

`XDataSource`

MATLAB variable, as a string

Link `XData` *to MATLAB variable*. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `XData`. The default value is an empty array.

```
set(h,'XDataSource','xdatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `XDataSource` does not change the object's `XData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`YData`

vector | matrix

Y-coordinates. A vector of *y*-coordinates defining the values along the *y*-axis for the graph. XData and ZData must be the same length and have the same number of rows.

YDataSource
MATLAB variable, as a string

Link YData *to MATLAB variable*. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the YData. The default value is an empty array.

```
set(h,'YDataSource','Ydatavariablename')
```

MATLAB requires a call to refreshdata when you set this property. Changing workspace variables used as an object's YDataSource does not change the object's YData values, but you can use refreshdata to force an update of the object's data. refreshdata also lets you specify that the data source variable be evaluated in the workspace of a function from which you call refreshdata.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

ZData
vector

Z-coordinates. A vector defining the *z*-coordinates for the graph. XData and YData must be the same length and have the same number of rows.

ZDataSource
MATLAB variable, as a string

Link `ZData` *to MATLAB variable.* Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `ZData`. The default value is an empty array.

```
set(h,'ZDataSource','zdatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `ZDataSource` does not change the object's `ZData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Purpose

Line specification string syntax

Description

Plotting functions accept *string specifiers* as arguments and modify the graph generated accordingly. Three components can be specified in the *string specifiers* along with the plotting command. They are:

- Line style
- Marker symbol
- Color

For example:

```
plot(x,y,'-.or')
```

plots y versus x using a dash-dot line (-.), places circular markers (o) at the data points, and colors both line and marker red (r). Specify the components (in any order) as a quoted string after the data arguments. Note that linespecs are single strings, not property-value pairs.

Plotting Data Points with No Line

If you specify a marker, but not a line style, only the markers are plotted. For example:

```
plot(x,y,'d')
```

Line Style Specifiers

You indicate the line styles, marker types, and colors you want to display using *string specifiers*, detailed in the following tables:

Specifier	LineStyle
'-'	Solid line (default)
'--'	Dashed line

Specifier	LineStyle
’:’	Dotted line
’-.’	Dash-dot line

Marker Specifiers

Specifier	Marker Type
’+’	Plus sign
’o’	Circle
’*’	Asterisk
’.’	Point
’x’	Cross
’square’ or ’s’	Square
’diamond’ or ’d’	Diamond
’^’	Upward-pointing triangle
’v’	Downward-pointing triangle
’>’	Right-pointing triangle
’<’	Left-pointing triangle
’pentagram’ or ’p’	Five-pointed star (pentagram)
’hexagram’ or ’h''’	Six-pointed star (hexagram)

Note The point (.) marker type does not change size when the specified value is less than 5.

Color Specifiers

Specifier	Color
`r`	Red
`g`	Green
`b`	Blue
`c`	Cyan
`m`	Magenta
`y`	Yellow
`k`	Black
`w`	White

Related Properties

This page also describes how to specify the properties of lines used for plotting. MATLAB graphics give you control over these visual characteristics:

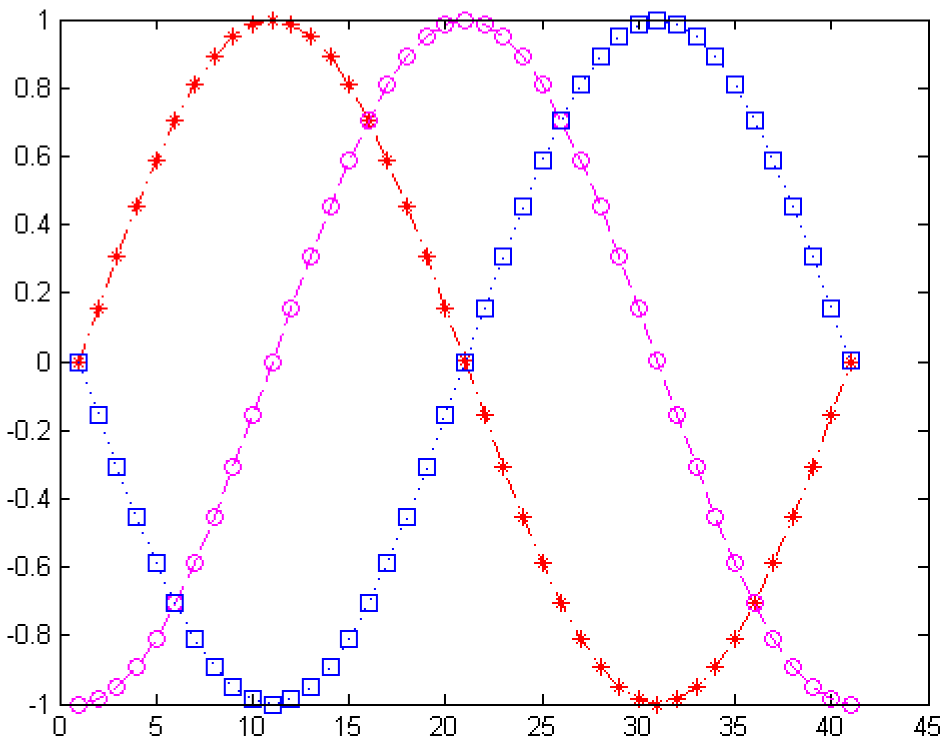
- `LineWidth` — Specifies the width (in points) of the line.
- `MarkerEdgeColor` — Specifies the color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).
- `MarkerFaceColor` — Specifies the color of the face of filled markers.
- `MarkerSize` — Specifies the size of the marker in points (must be greater than 0).

In addition, you can specify the `LineStyle`, `Color`, and `Marker` properties instead of using the symbol string. This is useful if you want to specify a color that is not in the list by using RGB values. See Lineseries Properties for details on these properties and `ColorSpec` for more information on color.

Examples

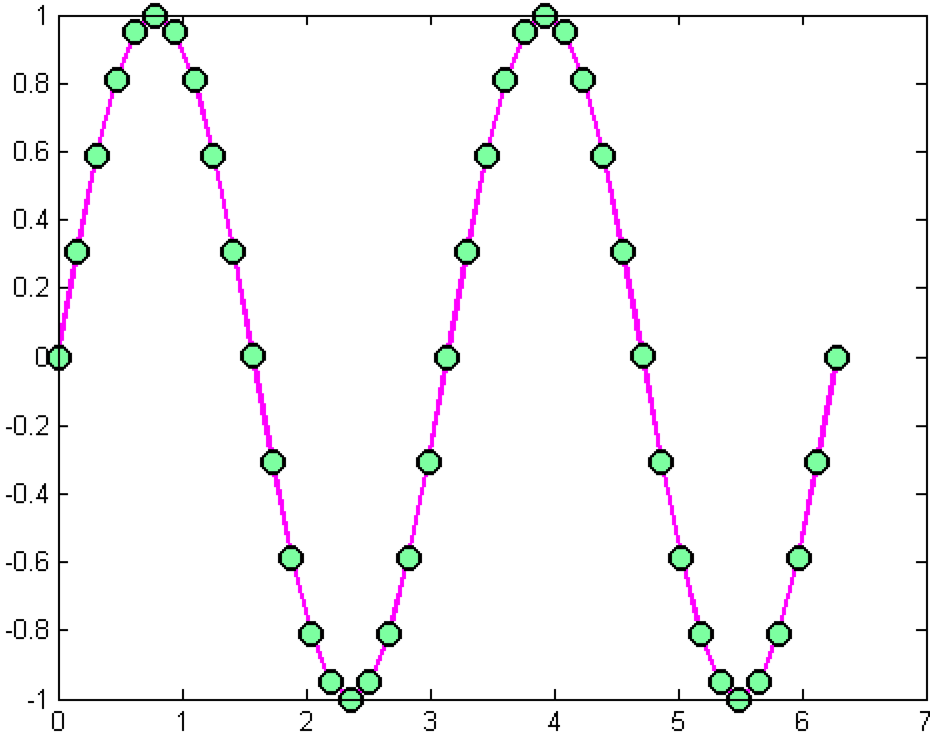
Plot the sine function over three different ranges using different line styles, colors, and markers.

```
figure
t = 0:pi/20:2*pi;
plot(t,sin(t),'-.r*')
hold on
plot(t,sin(t-pi/2),'--mo')
plot(t,sin(t-pi),':bs')
hold off
```



Create a plot illustrating how to set line properties:

```
figure
plot(t,sin(2*t),'-mo',...
                'LineWidth',2,...
                'MarkerEdgeColor','k',...
                'MarkerFaceColor',[.49 1 .63],...
                'MarkerSize',10)
```



See Also

axes | line | plot | patch | set | surface | ColorSpec | Lineseries Properties

linkaxes

Purpose

Synchronize limits of specified 2-D axes

Syntax

```
linkaxes(axes_handles)
linkaxes(axes_handles,'option')
```

Description

Use `linkaxes` to synchronize the individual axis limits across several figures or subplots within a figure. Calling `linkaxes` makes all input axes have identical limits. Linking axes is best when you want to zoom or pan in one subplot and display the same range of data in another subplot.

`linkaxes(axes_handles)` links the *x*- and *y*-axis limits of the axes specified in the vector `axes_handles`. You can link any number of existing plots or subplots. The `axes_handles` input should be a vector of the handles for each plot or subplot. Entering an array of values results in an error message.

`linkaxes(axes_handles,'option')` links the axes' `axes_handles` according to the specified option. The *`option`* argument can be one of the following strings:

`x`	Link *x*-axis only.
`y`	Link *y*-axis only.
`xy`	Link *x*-axis and *y*-axis.
`off`	Remove linking.

See the `linkprop` function for more advanced capabilities that allow you to link object properties on any graphics object.

Tips

The first axes you supply to `linkaxes` determines the *x*- and *y*-limits for all linked axes. This can cause plots to partly or entirely disappear if their limits or scaling are very different. To override this behavior, after calling `linkaxes`, specify the limits of the axes that you want to control with the `set` command, as the third example illustrates.

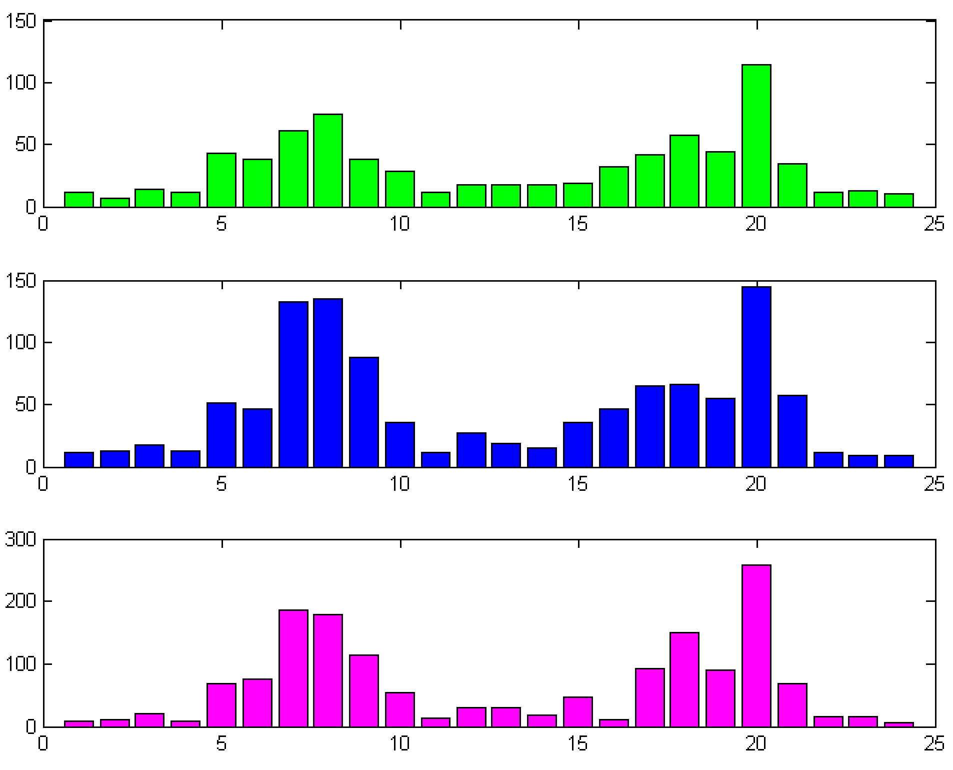
Note `linkaxes` is not designed to be transitive across multiple invocations. If you have three axes, `ax1`, `ax2`, and `ax3` and want to link them together, call `linkaxes` with `[ax1, ax2, ax3]` as the first argument. Linking `ax1` to `ax2`, then `ax2` to `ax3`, "unbinds" the `ax1`-`ax2` linkage.

Examples

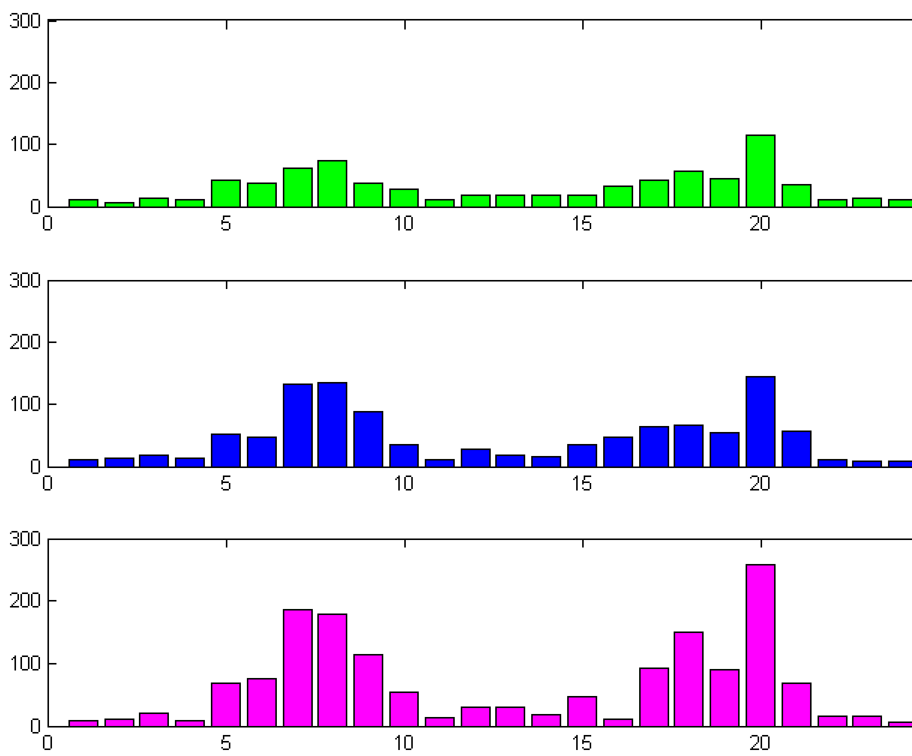
You can use interactive zooming or panning (selected from the figure toolbar) to see the effect of axes linking. For example, pan in one graph and notice how the *x*-axis also changes in the other. The axes responds in the same way to `zoom` and `pan` directives you type in the Command Window.

This example loads three vectors of data and creates a subplot for each vector. After a three-second pause, it then calls `linkaxes` with the argument `'xy'` to synchronize both the *x* and *y* axes limits. By specifying the third subplot first to `linkaxes`, you establish its y-limit for all the subplots:

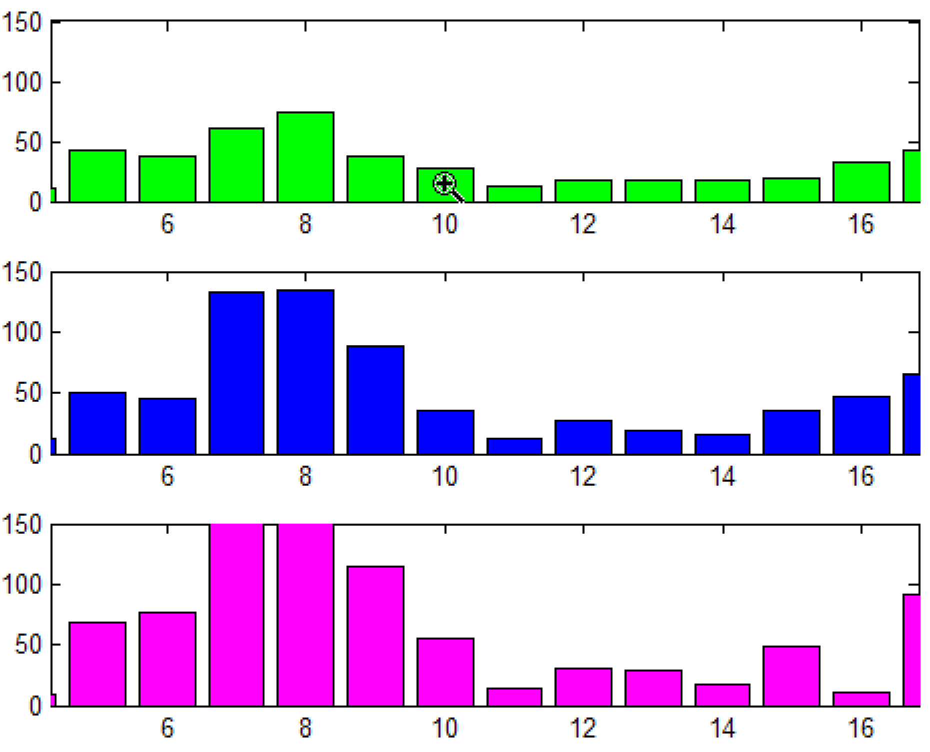
```
load count.dat  % Contains a 3-column vector named count
figure
ax(1) = subplot(3,1,1);
bar(ax(1),count(:,1),'g');
ax(2) = subplot(3,1,2);
bar(ax(2),count(:,2),'b');
ax(3) = subplot(3,1,3);
bar(ax(3),count(:,3),'m');
```




```
% Wait three seconds and then link the axes
pause(3)
linkaxes([ax(3) ax(2) ax(1)],'xy');  % Base Y-limits on bottom subplot
```



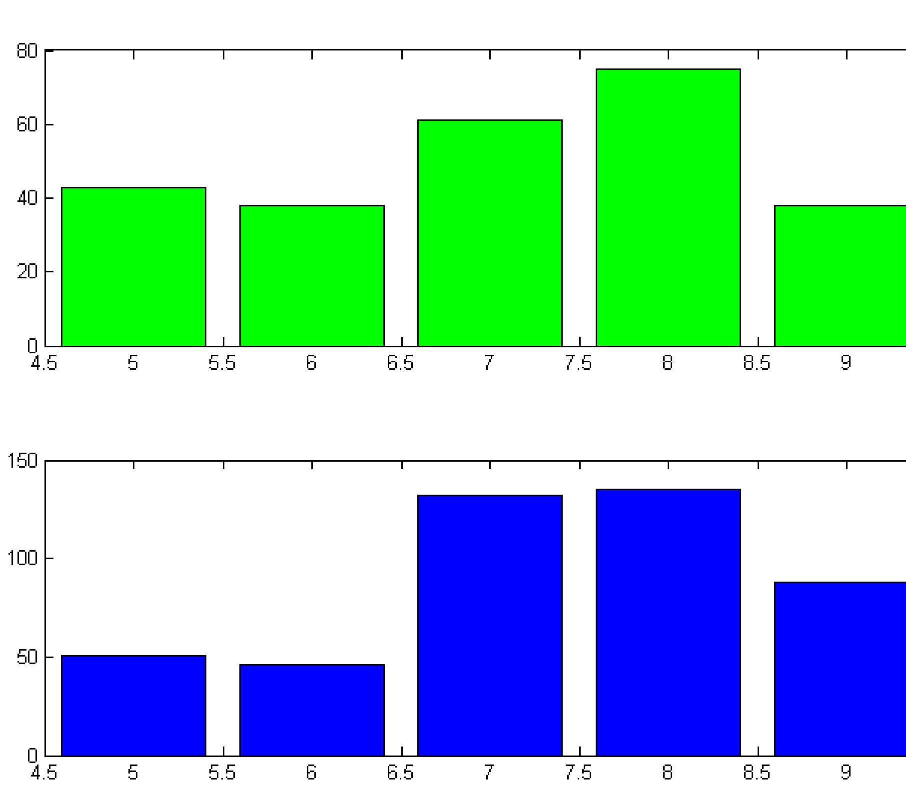
Click the **Zoom in** icon to enter Zoom mode, and click the mouse near the middle of the top axes. You observe all three axes respond the same. If you pan in any of the axes, all of them also respond the same.



Create two subplots containing related bar graphs. Call `linkaxes` to link only the *x*-axis limits of the two axes. Unlike in the previous example, the *y*-limits of the graphs remain unchanged. The example shows the effect of restricting the *x*-values to a range of 5 units and then manually panning either subplot:

```
load count.dat
figure
ax(1) = subplot(2,1,1);
```

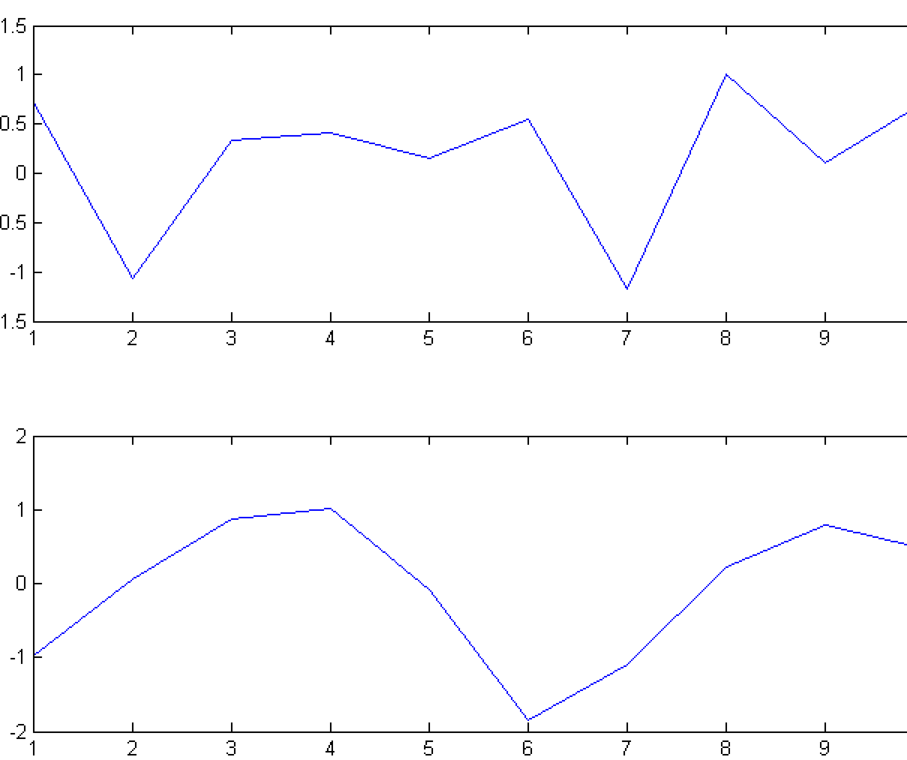
```
bar(ax(1),count(:,1),'g');
ax(2) = subplot(2,1,2);
bar(ax(2),count(:,2),'b');
linkaxes(ax,'x');
set(ax(1),'XLim',[4.5 9.5])    % Restrict either axis to show 5 values
```



Choose the Pan tool (**Tools > Pan**) (or type `pan on`). Drag the top axes. Both axes pan uniformly in *x*, but only the one you pan moves in the *y* direction.

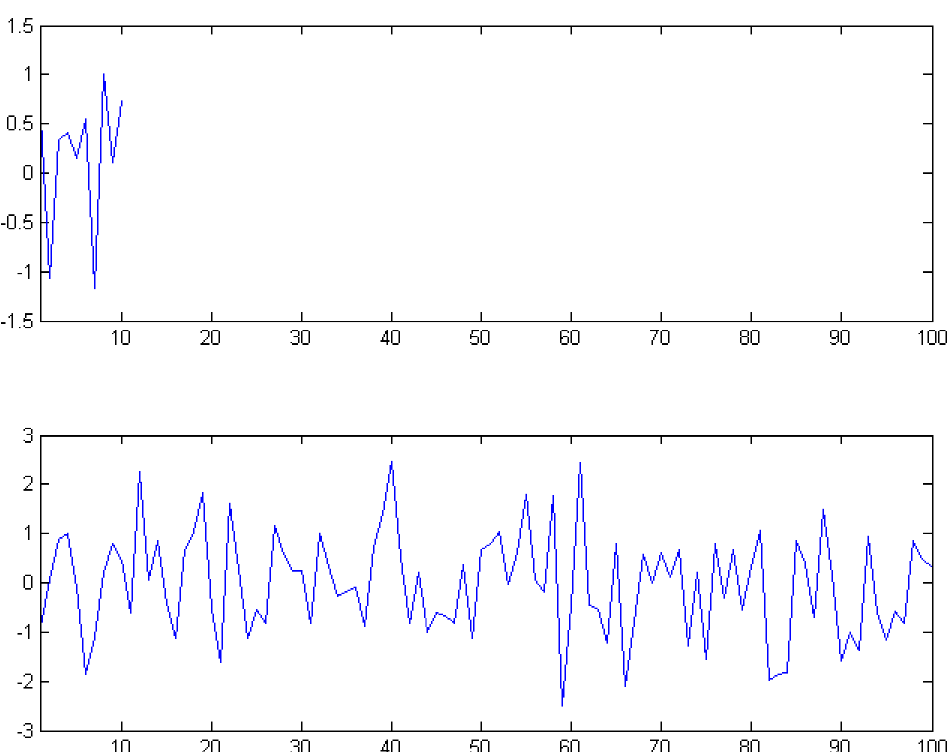
Create two subplots containing data having different ranges. The first axes handle passed to `linkaxes` determines the data range for all other linked axes. In this example, calling `set` for the lower axes overrides the *x*-limits established by the call to `linkaxes`:

```
a1 = subplot(2,1,1);
plot(randn(10,1));         % Plot 10 numbers on top
a2 = subplot(2,1,2);
plot(a2,randn(100,1))      % Plot 100 numbers below
linkaxes([a1 a2], 'x');    % Link the axes; subplot 2 now out of range
```



In order to display the full range of *x*-values, override the axes limits that `linkaxes` established.

```
set(a2,'xlimmode','auto'); % Now both axes run from 1-100 in x
                           % You could also use set(a2,'xlim',[1 100])
```



See Also

`linkdata` | `linkprop` | `pan` | `zoom`

Purpose

Automatically update graphs when variables change

Syntax

```
linkdata on
linkdata off
linkdata
linkdata(figure_handle,...)
linkobj = linkdata(figure_handle)
```

Description

`linkdata on` turns on data linking for the current figure.

`linkdata off` turns data linking off.

`linkdata` by itself toggles the state of data linking.

`linkdata(figure_handle,...)` applies the function to the specified figure handle.

`linkobj = linkdata(figure_handle)` returns a *linkdata object* for the specified figure. The object has one read-only property, `Enable`, the string `'on'` or `'off'`, depending on the linked state of the figure.

Data linking connects graphs in figure windows to variables in the base or a function's workspace via their `XDataSource`, `YDataSource`, and `ZDataSource` properties. When you turn on data linking for a figure, MATLAB compares variables in the current (base or function caller) workspace with the `XData`, `YData`, and `ZData` properties of graphs in the affected figure to try to match them. When a match is found, the appropriate `XDataSource`, `YDataSource` and/or `ZDataSource` for the graph are set to strings that name the matching variables.

Any subsequent changes to linked variables are reflected in graphs that use them as data sources and in the Variables editor, if the linked variables are displayed there. Conversely, any changes to plotted data values made at the command line, in the Variables editor, or with the Brush tool (such as deleting or replacing data points), are immediately reflected in the workspace variables linked to the data points.

When a figure containing graphs is linked and any variable identified as `XDataSource`, `YDataSource`, and/or `ZDataSource` changes its values in the workspace, all graphs displaying it in that and other linked figures

automatically update. This operation is equivalent to automatically calling the `refreshdata` function on the corresponding figure when a variable changes.

Linked figure windows identify themselves by the appearance of the Linked Plot information bar at the top of the window. When `linkdata` is `off` for a figure, the Linked Plot information bar is removed. If `linkdata` cannot unambiguously identify data sources for a graph in a linked figure, it reports this via the Linked Plot information bar, which gives the user an opportunity to identify data sources. The information bar displays a warning icon and a message, **No graphics have data sources** and also prompts **fix it**. Clicking **fix it** opens the Specify Data Source Properties dialog box for identifying variable names and ranges of data sources used in the graph.

Tips

- "Types of Variables You Can Link" on page 1-3092
- "Restoring Links that Break" on page 1-3092
- "Linking Rapidly Changing Data" on page 1-3093
- "Linking Brushed Graphs" on page 1-3093

Types of Variables You Can Link

You can use `linkdata` to connect a graph with scalar, vector and matrix numeric variables of any class (including `complex`, if the graphing function can plot it) — essentially any data for which `isnumeric` equals `true`. See "Example 3" on page 1-3095 for instructions on linking complex variables. You can also link plots to numeric fields within structures. You can specify MATLAB expressions as data sources, for example, `sqrt(y)+1`.

Restoring Links that Break

Refreshing data on a linked plot fails if the strings in the `XDataSource`, `YDataSource`, or `ZDataSource` properties, when evaluated, are incompatible with what is in the current workspace, such that the corresponding `XData`, `YData`, or `ZData` are unable to respond. The visual appearance of the object in the graph is not affected by such failures, so graphic objects show no indication of broken links. Instead,

a warning icon and the message **Failing links** appear on the Linked Plot information bar along with an **Edit** button that opens the Specify Data Sources dialog box.

Linking Rapidly Changing Data

`linkdata` buffers updates to data and dispatches them to plots at roughly half-second intervals. This makes data linking not suitable for smoothly animating changes in data values unless they are updated in loops that are forced to execute two times per second or less.

One consequence of buffering link updates is that `linkdata` might not detect changes in data streams it monitors. If you are running a function that uses `assignin` or `evalin` to update workspace variables, `linkdata` can sometimes fail to process updates that change values but not the size and class of workspace variables. Such failures only happen when the function itself updates the plot.

Linking Brushed Graphs

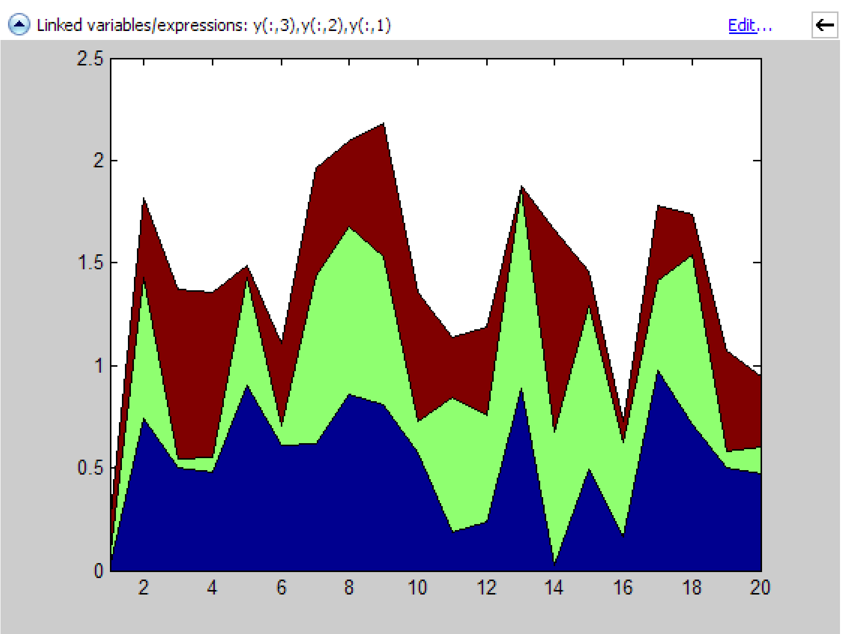
If you link data sources to graphs that have been brushed, their brushing marks can change or vanish. This is because the workspace variables in those graphs now dictate which, if any, observations are brushed, superseding any brushing annotations that were applied to their graphical data (`YData`, etc.). For more details, see "How Data Linking Affects Data Brushing" on page 1-551 in the `brush` reference page.

Examples

Example 1

Create two variables, graph them as areaseries, and link the plot to them:

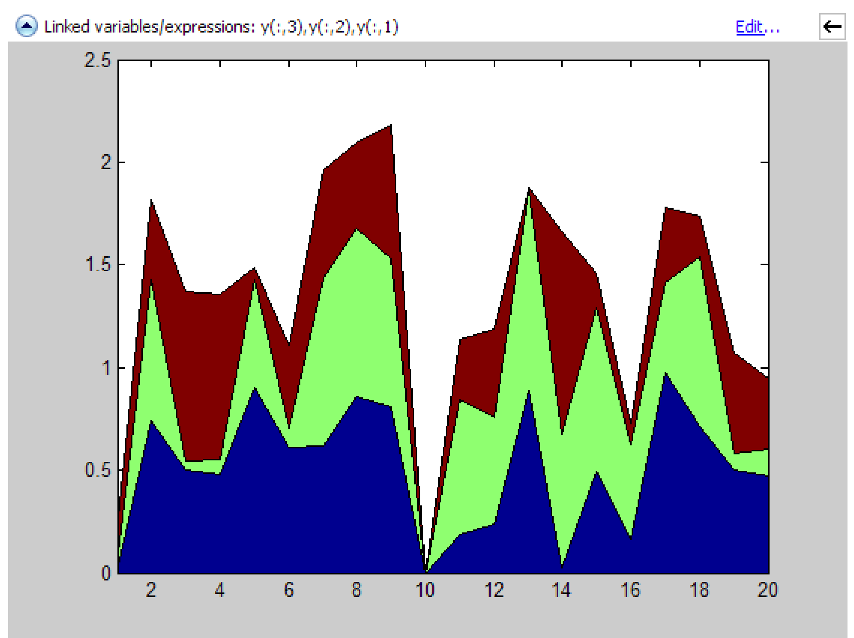
```
x = [1:20];
y = rand(20,3);
area(x,y)
linkdata on
```



Change values for linked variable `y` in the workspace:

```
y(10,:) = 0;
```

The area graph immediately updates.



Example 2

Delete a figure if it is not linked, based on a returned `linkdata` object:

```
ld = linkdata(fig)

ld =
     graphics.linkdata

if strcmp(ld.Enable,'off')
    delete(fig)
end
```

Example 3

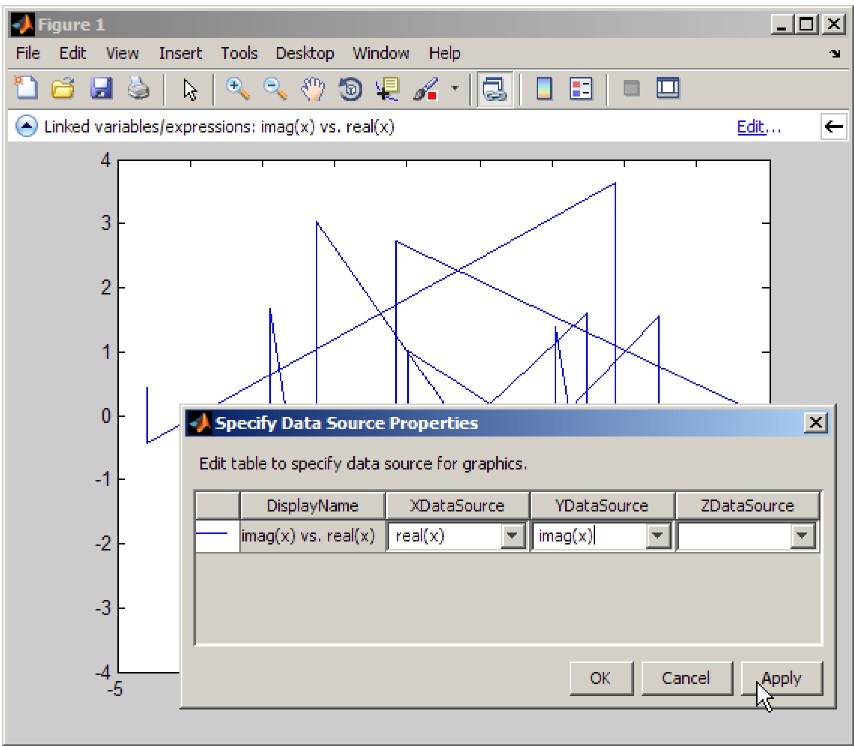
If a graphing function can display a complex variable, you can link such plots. To do so, you need to describe the data sources as expressions to separate the real and imaginary parts of the variable. For example,

```
x = eig(randn(20,20))
whos
  Name        Size      Bytes  Class     Attributes
  x          20x1         320  double    complex
```

yields a complex vector. You can use `plot` to display the real portion as *x* and the imaginary portion as *y*, then link the graph to the variable:

```
plot(x)
linkdata
```

However, `linkdata` cannot unambiguously identify the graph's data sources, and you must tell it by typing `real(x)` and `imag(x)` into the Specify Data Source Properties dialog box that displays when you click **fix it** in the Linked Plot information bar.



To avoid having to type the data source names in the dialog box, you can specify them when you plot:

```
plot(x,'XDataSource','real(x)','YDataSource','imag(x)')
```

If you subsequently change values of x programmatically or manually, the plot updates accordingly.

Note Although you can use data brushing on linked plots of complex data, your brush marks only appear in the plot you are brushing, not in other plots or in the Variables editor. This is because function calls, such as `real(x)` and `imag(x)`, that you specify as data sources are not interpreted when brushing graphed data.

See Also

`brush` | `linkaxes` | `linkprop` | `refreshdata`

How To

- "Making Graphs Responsive with Data Linking"

linkprop

Purpose

Keep same value for corresponding properties of graphics objects

Syntax

```
hlink = linkprop(obj_handles,'PropertyName')
hlink = linkprop(obj_handles,{'PropertyName1','PropertyName2',...})
```

Description

Use `linkprop` to maintain the same values for the corresponding properties of different graphics objects.

Note Use `linkprop` only with Handle Graphics objects.

`hlink = linkprop(obj_handles,'PropertyName')` maintains the same value for the property *`PropertyName`* on all objects whose handles appear in `obj_handles`. `linkprop` returns the link object in `hlink`. See "About Link Objects" on page 1-3100 for more information.

`hlink = linkprop(obj_handles,{'PropertyName1','PropertyName2',...})` maintains the same respective values for all properties passed as a cell array on all objects whose handles appear in `obj_handles`.

MATLAB updates the linked properties of all linked objects immediately when `linkprop` is called. The first object in the list `obj_handles` determines the property values for the other objects.

A set of graphics objects can have only one link object connecting their properties at any given time. Calling `linkprop` creates a new link object. This new link object replaces any existing link object that is associated with the objects specified in `obj_handles`. However, you can manage which properties and which objects are linked by calling methods on that object:

- To add an object to the list of linked objects, use the `addtarget` method.
- To link new properties of currently-linked objects, use the `addprop` method.
- To stop linking an object, use the `removetarget` method.

- To stop properties from linking, use the removeprop method.

About Link Objects

The link object that linkprop returns stores the mechanism that links the properties of different graphics objects. Therefore, the link object must exist within the context where you want property linking to occur (such as in the base workspace if users are to interact with the objects from the command line or figure tools).

The following list describes ways to maintain a reference to the link object.

- Return the link object as an output argument from a function and keep it in the base workspace while interacting with the linked objects.
- Make the hlink variable global.
- Store the hlink variable in an object's UserData property or in application data. See the "Examples" on page 1-3101 section for an example that uses application data.

Updating a Link Object

If you want to change either the graphics objects or the properties that are linked, you need to use the link object methods designed for that purpose. These methods are functions that operate only on link objects. To use them, you must first create a link object using linkprop.

Method	Purpose
addtarget	Add specified graphics object to the link object's targets.
removetarget	Remove specified graphics object from the link object's targets.
addprop	Add specified property to the linked properties.
removeprop	Remove specified property from the linked properties.

Method Syntax

```
addtarget(hlink,obj_handles)
```

```
removetarget(hlink,obj_handles)
addprop(hlink,'PropertyName')
removeprop(hlink,'PropertyName')
```

Method Arguments

- `hlink` — Link object returned by `linkprop`
- `obj_handles` — One or more graphic object handles
- *`PropertyName`* — Name of a property common to all target objects

Examples

This example creates four isosurface graphs of fluid flow data, each displaying a different isovalue. The `CameraPosition` and `CameraUpVector` properties of each subplot axes are linked so that the user can rotate all subplots in unison.

After running the example, select **Rotate 3D** from the figure **Tools** menu and observe how all subplots rotate together.

Note You can run this example or open it in the MATLAB editor.

The property linking code is in step 3.

1. Define the data using `flow` and specify property values for the isosurface (which is a patch object).

```
[x,y,z,v] = flow;
isoval = [-3,-1,0,1];
props.FaceColor = [0,0,0.5];
props.EdgeColor = 'none';
props.AmbientStrength = 1;
props.FaceLighting = 'gouraud';
```

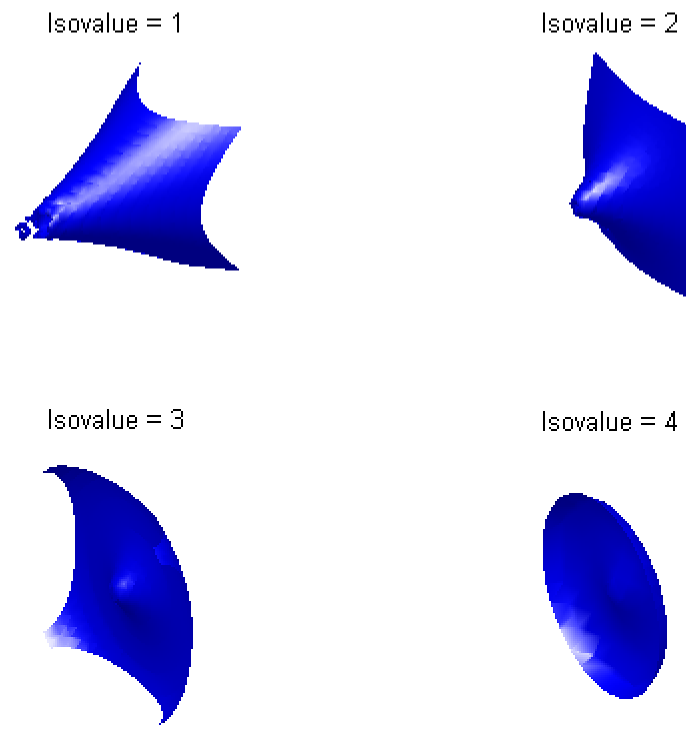
2. Create four subplot axes and add an isosurface graph to each one. Add a title and set viewing and lighting parameters. MATLAB functions used are `subplot`, `patch`, `isosurface`, `title`, and `num2str`.

Also set the viewing and lighting parameters for each axes to be the same by calling `view`, `axis`, and `camlight`.

```
figure
% Preallocate h
h = zeros(1,4);
for k = 1:4
   h(k) = subplot(2,2,k);
   patch(isosurface(x,y,z,v,isoval(k)),props)
   title(h(k),['Isovalue = ',num2str(k)])
   % Set the view and add lighting
   view(h(k),3); axis(h(k),'tight','equal')
   camlight left; camlight right
   % Make axes invisible and title visible
   axis(h(k),'off')
   set(get(h(k),'title'),'Visible','on')
end
```

3 Link the `CameraPosition` and `CameraTarget` properties of all subplot axes. Since this code already js completed executing when you rotate the subplots, the link object is stored in the first subplot axes application data. See `setappdata` for more information on using application data.

```
if ishghandle(h)
   hlink = linkprop(h,{'CameraPosition','CameraUpVector'});
   key = 'graphics_linkprop';
   % Store link object on first subplot axes
   setappdata(h(1),key,hlink);
end
```



4 Turn on the **Rotate 3D** tool. Click any object in the figure and rotate it. As you turn it, all three other subplots turn in the same manner.

```
rotate3d on
```

Linking an Additional Property

Suppose you want to add the axes `PlotBoxAspectRatio` to the linked properties in the previous example. You can do this by modifying the link object that is stored in the first subplot axes' application data.

1. First click the first subplot axes to make it the current axes (since its handle was saved only within the creating function). Then get the link object's handle from application data (`getappdata`).

   ```
   hlink = getappdata(gca,'graphics_linkprop');
   ```

2. Use the `addprop` method to add a new property to the link object.

   ```
   addprop(hlink,'PlotBoxAspectRatio')
   ```

Since `hlink` is a reference to the link object (i.e., not a copy), `addprop` can change the object that is stored in application data.

See Also

`getappdata` | `ishghandle` | `linkaxes` | `linkdata` | `setappdata`

Purpose

Solve linear system of equations

Syntax

```
X = linsolve(A,B)
X = linsolve(A,B,opts)
```

Description

`X = linsolve(A,B)` solves the linear system `A*X = B` using LU factorization with partial pivoting when A is square and QR factorization with column pivoting otherwise. The number of rows of `A` must equal the number of rows of `B`. If `A` is m-by-n and `B` is m-by-k, then `X` is n-by-k. `linsolve` returns a warning if `A` is square and ill conditioned or if it is not square and rank deficient.

`[X, R] = linsolve(A,B)` suppresses these warnings and returns `R`, which is the reciprocal of the condition number of `A` if `A` is square, or the rank of `A` if `A` is not square.

`X = linsolve(A,B,opts)` solves the linear system `A*X = B` or `A'*X = B`, using the solver that is most appropriate given the properties of the matrix `A`, which you specify in `opts`. For example, if `A` is upper triangular, you can set `opts.UT = true` to make `linsolve` use a solver designed for upper triangular matrices. If `A` has the properties in `opts`, `linsolve` is faster than `mldivide`, because `linsolve` does not perform any tests to verify that `A` has the specified properties.

Notes If `A` does not have the properties that you specify in `opts`, `linsolve` returns incorrect results and does not return an error message. If you are not sure whether `A` has the specified properties, use `mldivide` instead.

For small problems, there is no speed benefit in using `linsolve` on triangular matrices as opposed to using the `mldivide` function.

The `TRANSA` field of the `opts` structure specifies the form of the linear system you want to solve:

- If you set `opts.TRANSA = false`, `linsolve(A,B,opts)` solves `A*X = B`.
- If you set `opts.TRANSA = true`, `linsolve(A,B,opts)` solves `A'*X = B`.

The following table lists all the field of `opts` and their corresponding matrix properties. The values of the fields of `opts` must be `logical` and the default value for all fields is `false`.

Field Name	Matrix Property
LT	Lower triangular
UT	Upper triangular
UHESS	Upper Hessenberg
SYM	Real symmetric or complex Hermitian
POSDEF	Positive definite
RECT	General rectangular
TRANSA	Conjugate transpose — specifies whether the function solves `A*X = B` or `A'*X = B`

The following table lists all combinations of field values in `opts` that are valid for `linsolve`. A true/false entry indicates that `linsolve` accepts either true or false.

LT	UT	UHESS	SYM	POSDEF	RECT	TRANSA
true	false	false	false	false	true/false	true/false
false	true	false	false	false	true/false	true/false
false	false	true	false	false	false	true/false
false	false	false	true	true/false	false	true/false
false	false	false	false	false	true/false	true/false

Examples

The following code solves the system `A'x = b` for an upper triangular matrix `A` using both `mldivide` and `linsolve`.

```
A = triu(rand(5,3)); x = [1 1 1 0 0]'; b = A'*x;
y1 = (A')\b
opts.UT = true; opts.TRANSA = true;
y2 = linsolve(A,b,opts)

y1 =

    1.0000
    1.0000
    1.0000
         0
         0


y2 =

    1.0000
    1.0000
    1.0000
         0
         0
```

Note If you are working with matrices having different properties, it is useful to create an options structure for each type of matrix, such as `opts_sym`. This way you do not need to change the fields whenever you solve a system with a different type of matrix `A`.

See Also

`mldivide`

linspace

Purpose

Generate linearly spaced vectors

Syntax

```
y = linspace(a,b)
y = linspace(a,b,n)
```

Description

The `linspace` function generates linearly spaced vectors. It is similar to the colon operator ":", but gives direct control over the number of points.

`y = linspace(a,b)` generates a row vector `y` of 100 points linearly spaced between and including `a` and `b`.

`y = linspace(a,b,n)` generates a row vector `y` of `n` points linearly spaced between and including `a` and `b`. For `n < 2`, `linspace` returns `b`.

Examples

Create a vector of 100 linearly spaced numbers from 1 to 500:

```
A = linspace(1,500);
```

Create a vector of 12 linearly spaced numbers from 1 to 36:

```
A = linspace(1,36,12);
```

See Also

`logspace` | `colon operator`

Purpose Random number generator algorithms

Class `RandStream`

Syntax `RandStream.list`

Description

`RandStream.list` lists all the generator algorithms that may be used when creating a random number stream with `RandStream` or `RandStream.create`. The available generator algorithms and their properties are given in the following table.

Keyword	Generator	Multiple Stream and Substream Support	Approximate Period In Full Precision
mt19937ar	Mersenne twister (used by default stream at MATLAB startup)	No	$2^{19937}-1$
mcg16807	Multiplicative congruential generator	No	$2^{31}-2$
mlfg6331_64	Multiplicative lagged Fibonacci generator	Yes	2^{124}
mrg32k3a	Combined multiple recursive generator	Yes	2^{127}

Keyword	Generator	Multiple Stream and Substream Support	Approximate Period In Full Precision
shr3cong	Shift-register generator summed with linear congruential generator	No	2^{64}
swb2712	Modified subtract with borrow generator	No	2^{1492}

See “Choosing a Random Number Generator” for details about these generator algorithms. See http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html for a full description of the Mersenne twister algorithm.

Purpose

Create and open list-selection dialog box

Syntax

```
[Selection,ok] = listdlg('ListString',S)
```

Description

`[Selection,ok] = listdlg('ListString',S)` creates a modal dialog box that enables you to select one or more items from a list. `Selection` is a vector of indices of the selected strings (in single selection mode, its length is 1). `Selection` is `[]` when `ok` is 0. `ok` is 1 if you click the **OK** button, or 0 if you click the **Cancel** button or close the dialog box. Double-clicking on an item or pressing **Return** when multiple items are selected has the same effect as clicking the **OK** button. The dialog box has a **Select all** button (when in multiple selection mode) that enables you to select all list items.

Inputs are in parameter/value pairs:

Parameter	Description
`'ListString'`	Cell array of strings that specify the list box items.
`'SelectionMode'`	String indicating whether one or many items can be selected: `'single'` or `'multiple'` (the default).
`'ListSize'`	List box size in pixels, specified as a two-element vector `[width height]`. Default is `[160 300]`.
`'InitialValue'`	Vector of indices of the list box items that are initially selected. Default is 1, the first item.
`'Name'`	String for the dialog box's title. Default is ''.
`'PromptString'`	String matrix or cell array of strings that appears as text above the list box. Default is {}.
`'OKString'`	String for the OK button. Default is `'OK'`.
`'CancelString'`	String for the Cancel button. Default is '`Cancel`'.
`'uh'`	Uicontrol button height, in pixels. Default is 18.

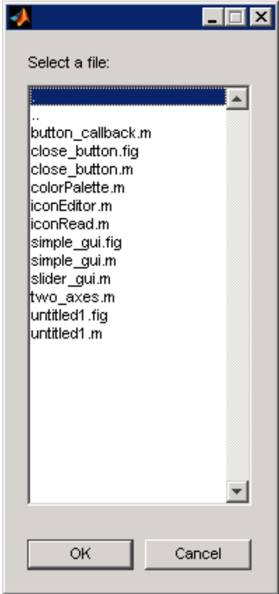
Parameter	Description
'fus'	Frame/uicontrol spacing, in pixels. Default is 8.
'ffs'	Frame/figure spacing, in pixels. Default is 8.

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

Examples

This example displays a dialog box that enables the user to select a file from the current directory. The function returns a vector. Its first element is the index to the selected file; its second element is 0 if no selection is made, or 1 if a selection is made.

```
d = dir;
str = {d.name};
[s,v] = listdlg('PromptString','Select a file:',...
                'SelectionMode','single',...
                'ListString',str)
```



See Also

`dialog` | `errordlg` | `helpdlg` | `inputdlg` | `msgbox` | `questdlg` | `warndlg` | `dir` | `figure` | `uiwait` | `uiresume`

listfonts

Purpose

List available system fonts

Syntax

```
c = listfonts
c = listfonts(h)
```

Description

`c = listfonts` returns sorted list of available system fonts.

`c = listfonts(h)` returns sorted list of available system fonts and includes the `FontName` property of the object with handle `h`.

Tips

Calling `listfonts` returns a list of all fonts residing on your system, possibly including fonts that cannot be used because they are bitmapped. You can instead use the `uisetfont` utility (a GUI) to preview fonts you might want to use; it only displays fonts that can be rendered in MATLAB figures and GUIs. Like `uisetfont`, the **Custom Fonts** pane of MATLAB Preferences also previews available fonts and only shows those that can be rendered.

Examples

Example 1

This example returns a list of available system fonts similar in format to the one shown.

```
list = listfonts

list =
    'Agency FB'
    'Algerian'
    'Arial'
    ...
    'ZapfChancery'
    'ZapfDingbats'
    'ZWAdobeF'
```

Example 2

This example returns a list of available system fonts with the value of the `FontName` property, for the object with handle `h`, sorted into the list.

```
 h = uicontrol('Style','text','String',...
     'My Font','FontName','MyFont');
 list = listfonts(h)

list =
    'Agency FB'
    'Algerian'
    'Arial'
    ...
    'MyFont'
    ...
    'ZapfChancery'
    'ZapfDingbats'
    'ZWAdobeF'
```

See Also

uisetfont

load

Purpose

Load data from MAT-file into workspace

Syntax

```
S = load(filename)
S = load(filename, variables)
S = load(filename, '-mat', variables)
S = load(filename, '-ascii')
load(filename, ___)
load filename ___
```

Description

`S = load(filename)` loads the variables from a MAT-file into a structure array, or data from an ASCII file into a double-precision array.

`S = load(filename, variables)` loads the specified variables from a MAT-file.

`S = load(filename, '-mat', variables)` forces `load` to treat the file as a MAT-file, regardless of the extension. Specifying *`variables`* is optional.

`S = load(filename, '-ascii')` forces `load` to treat the file as an ASCII file, regardless of the extension.

`load(filename, ___)` loads without combining MAT-file variables into a structure array.

`load filename ___` is the command form of the syntax, for convenient loading from the command line. With command syntax, you do not need to enclose input strings in single quotation marks. Separate inputs with spaces instead of commas. Do not use command syntax if inputs such as *`filename`* are variables. For more information, see "Command vs. Function Syntax" in the MATLAB Programming Fundamentals documentation.

Input Arguments

filename

Name of a file. If you do not specify *`filename`*, the `load` function searches for a file named `matlab.mat`.

`filename` can include a file extension and a full or partial path. If *`filename`* has no extension (that is, no text after a period), `load` looks

for a file named *filename*.mat. If *filename* has an extension other than .mat, the load function treats the file as ASCII data.

Default: 'matlab.mat'

variables

List of variables to load from the file. Valid for MAT-files only. Use one of the following forms:

var1, var2, ...	Load the listed variables. Use the '*' wildcard to match patterns. For example, load('A*') loads all variables that start with A.
'-regexp', *expressions*	Load only the variables or fields that match the specified regular expressions.

Default: all variables in file

'-mat'

Keyword that indicates that the specified file is a MAT-file, regardless of the extension.

'-ascii'

Keyword that indicates that the specified file is an ASCII file, regardless of the extension.

The following remarks apply to loading all ASCII files, even when you do not include the -ascii keyword, but load a file with an extension other than .mat:

- The file must contain a rectangular table of numbers, with an equal number of elements in each row. The file delimiter (the character between elements in each row) can be a blank, comma, semicolon, or

tab character. The file can contain MATLAB comments (lines that begin with a percent sign, `%`).

- MATLAB returns the data as a single two-dimensional array of type `double`. The number of rows in the array is equal to the number of lines in the file. The number of columns is equal to the number of values on a line.
- If you do not specify an output for the `load` function, MATLAB creates a variable named after the loaded file (minus any file extension). For example, the command

  ```
  load mydata.dat
  ```

  reads data into a variable called `mydata`.

  To create the variable name, `load` precedes any leading underscores or digits in *`filename`* with an `X` and replaces any other nonalphabetic characters with underscores. For example, the command

  ```
  load 10-May-data.dat
  ```

  creates a variable called `X10_May_data`.

Examples

Load all variables from the example MAT-file `gong.mat`. Check the contents of the workspace before and after the load operation.

```
disp('Contents of workspace before loading file:')
whos

disp('Contents of gong.mat:')
whos -file gong.mat

load gong.mat
disp('Contents of workspace after loading file:')
whos
```

Load only the variable `y` from the example file `handel.mat`. If the workspace already contains a variable `y`, the `load` operation overwrites it with data from the file.

```
load('handel.mat', 'y')
```

Create a list of variables to load using a cell array. Call `load` using function syntax.

```
%This example file contains variables X, caption, and map.
exampleFile = 'durer.mat';

% Load selected variables.
vars = {'X', 'caption'};
load(exampleFile, vars{:});
```

Use regular expressions to load four variables from the example file `accidents.mat` that do not begin with `'hwy'`:

```
load('accidents.mat', '-regexp', '^(?!hwy)...');
```

Load the first three variables from `accidents.mat`, determined alphabetically by variable name:

```
vars = whos('-file', 'accidents.mat');
load('accidents.mat', vars(1:3).name);
```

Create an ASCII file from several 4-column matrices, and load the data back into a double array named `mydata`:

```
a = magic(4);
b = ones(2, 4) * -5.7;
```

```
c = [8 6 4 2];
save -ascii mydata.dat a b c
clear a b c

load mydata.dat
```

See Also

`clear` | `importdata` | `matfile` | `regexp` | `save` | `uiimport` | `whos`

How To

- “Supported File Formats”
- “Save, Load, and Delete Workspace Variables”
- “Ways to Import Text Files”
- “Troubleshooting: Loading Variables within a Function”
- “Process a Sequence of Files”

Purpose

Initialize control object from file

Syntax

```
h.load('filename')
load(h, 'filename')
```

Description

`h.load('filename')` initializes the COM object associated with the interface represented by the MATLAB COM object `h` from file specified in the string `filename`. The file must have been created previously by serializing an instance of the same control.

`load(h, 'filename')` is an alternate syntax for the same operation.

Note The COM `load` function is only supported for controls at this time.

Tips

COM functions are available on Microsoft Windows systems only.

Examples

Create an `mwsamp` control and save its original state to the file `mwsample`:

```
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f);
h.save('mwsample')
```

Now, alter the figure by changing its label and the radius of the circle:

```
h.Label = 'Circle';
h.Radius = 50;
h.Redraw;
```

Using the `load` function, you can restore the control to its original state:

```
h.load('mwsample');
h.get
```

MATLAB displays the original values:

```
ans =
```

```
 Label: 'Label'
Radius: 20
```

See Also

save (COM) | actxcontrol | actxserver | release | delete (COM)

Purpose

Load serial port objects and variables into MATLAB workspace

Syntax

```
load filename
load filename obj1 obj2...
```

Description

`load filename` returns all variables from the file specified by `filename` into the MATLAB workspace.

`load filename obj1 obj2...` returns the serial port objects specified by `obj1 obj2` ... from the file `filename` into the MATLAB workspace.

`out = load('filename','obj1','obj2',...)` returns the specified serial port objects from the file `filename` as a structure to `out` instead of directly loading them into the workspace. The field names in `out` match the names of the loaded serial port objects.

Tips

Values for read-only properties are restored to their default values upon loading. For example, the `Status` property is restored to `closed`. To determine if a property is read-only, examine its reference pages.

Examples

Note This example is based on a Windows platform.

Suppose you create the serial port objects `s1` and `s2`, configure a few properties for `s1`, and connect both objects to their instruments:

```
s1 = serial('COM1');
s2 = serial('COM2');
set(s1,'Parity','mark','DataBits',7);
fopen(s1);
fopen(s2);
```

Save `s1` and `s2` to the file `MyObject.mat`, and then load the objects back into the workspace:

```
save MyObject s1 s2;
load MyObject s1;
```

```
load MyObject s2;

get(s1, {'Parity', 'DataBits'})
ans =
    'mark'    [7]
get(s2, {'Parity', 'DataBits'})
ans =
    'none'    [8]
```

See Also

save | Status

Purpose

Load shared library into MATLAB

Syntax

```
loadlibrary(libname,hfile)
loadlibrary(libname,hfile,Name,Value)

loadlibrary(libname,@protofile)

[notfound,warnings] = loadlibrary( ___ )
```

Description

`loadlibrary(libname,hfile)` loads functions from shared library, `libname`, defined in header file, `hfile`, into MATLAB.

`loadlibrary(libname,hfile,Name,Value)` loads the library with one or more `Name,Value` arguments.

`loadlibrary(libname,@protofile)` uses a prototype file, `protofile`, in place of a header file.

`[notfound,warnings] = loadlibrary( ___ )` returns warning information, and can include any of the input arguments in previous syntaxes.

Limitations

- You must select a supported C compiler and Perl must be available. To select a compiler, use the `mex -setup` command, described in “Selecting a Compiler on Windows Platforms” or “Selecting a Compiler on UNIX Platforms”. For an up-to-date list of supported compilers, see the Supported and Compatible Compilers Web page.
- Do not call `loadlibrary` if the library is already in memory. To test this condition, call `libisloaded`.
- `loadlibrary` does not support libraries generated by the MATLAB Compiler™ product.
- The MATLAB Shared Library interface does not support library functions with function pointer inputs.

Input Arguments

libname - Name of shared library

string

Name of shared library, specified as a string. The name is case sensitive and must match the file on your system.

On Microsoft Windows systems, `libname` refers to the name of a shared library (`.dll`) file. On Linux systems, it refers to the name of a shared object (`.so`) file. On Apple Macintosh systems, it refers to a dynamic shared library (`.dylib`).

If you do not include a file extension with the `libname` argument, `loadlibrary` attempts to find the library with either the appropriate platform MEX-file extension or the appropriate platform library extension (usually `.dll`, `.so`, or `.dylib`). For a list of MEX-file extensions, see "Binary MEX-File Extensions".

Data Types

`char`

hfile - Name of C header file

string

Name of C header file, specified as a string. The name is case sensitive and must match the file on your system. If you do not include a file extension in the file name, `loadlibrary` uses `.h` for the extension.

Data Types

`char`

protofile - Name of prototype file

string

Name of prototype file, specified as a string. The name is case sensitive and must match the file on your system. The string `@protofile` specifies a function handle to the prototype file. When using a prototype file, the only valid `Name,Value` pair argument is **`alias`**.

Data Types

`char`

Name-Value Pair Arguments

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

Example:

'addheader' - Header file

string

Header file, specified as the comma-separated pair consisting of '`addheader`' and a string. Specify the file name without a file extension.

Each file specified by `addheader` must have a corresponding `#include` statement in the base header file. Use `addheader` to load only the functions defined in the header file that you want to use in MATLAB.

MATLAB does not verify the existence of header files and ignores any that are not needed.

'alias' - Alternative name for library

string

Alternative name for library, specified as the comma-separated pair consisting of '`alias`' and a string. Associates the specified name with the library. All subsequent calls to MATLAB functions that reference this library must use this alias until the library is unloaded.

'includepath' - Additional search path for header files

string

Additional search path for header files, specified as the comma-separated pair consisting of '`includepath`' and a string.

'mfilename' - Prototype file

string

Prototype file, specified as the comma-separated pair consisting of 'mfilename' and a string. Generates a prototype file in the current folder. The prototype file name must be different from the library name. Use this file in place of a header file when loading the library.

'thunkfilename' - Thunk file
string

Thunk file, specified as the comma-separated pair consisting of 'thunkfilename' and a string. Overrides the default thunk file name.

Output Arguments

notfound - Names of functions
cell array

Names of functions found in header files but missing from the library, returned as cell array.

Data Types
`cell`

warnings - Warnings
character array

Warnings produced while processing the header file, returned as character array.

Examples

Load Functions in shrlibsample Library

Add path to examples folder.

```
addpath(fullfile(matlabroot,'extern','examples','shrlib'))
```

Load `shrlibsample`.

```
loadlibrary('shrlibsample')
```

Clean up.

```
unloadlibrary shrlibsample
```

Load Library Using Header File

The header file for the `libmx` library is `matrix.h`.

```
hfile = fullfile(matlabroot,'extern','include','matrix.h');
loadlibrary('libmx',hfile)
```

Clean up.

```
unloadlibrary libmx
```

Load Library Using Multiple Header Files

Suppose you have a library, `mylib`, with the header file, `mylib.h`. The header file contains the statement, `#include header2.h`. To use functions defined in `header2.h`, call `loadlibrary` with the `addheader` option.

```
loadlibrary('mylib','mylib.h','addheader','header2')
```

Load Library Using an Alias Name

Create an alias, `lib`, for library, `shrlibsample`.

```
loadlibrary('shrlibsample','alias','lib')
```

Use the alias name, `lib`, to call a function in the library.

```
str = 'This was a Mixed Case string';
calllib('lib','stringToUpper',str)
```

```
ans =
   THIS WAS A MIXED CASE STRING
```

Clean up.

```
unloadlibrary lib
```

Search Alternative Paths for Header Files

Create an alternative path, `apath`, containing header files.

```
apath = fullfile(matlabroot,'extern','include');
```

Load `shrlibsample`.

```
loadlibrary('shrlibsample','includepath',apath);
```

Clean up.

```
unloadlibrary shrlibsample
```

Tips

- If you have more than one library file of the same name, load the first using the library file name, and load the additional libraries using the **`alias`** option.

Definitions

Prototype File

A prototype file is a file of MATLAB commands which you can modify and use in place of a header file.

Thunk File

A thunk file is a compatibility layer to a 64-bit library generated by MATLAB. The name of the thunk file is:

```
BASENAME_thunk_COMPUTER.c
```

where *`BASENAME`* is either the name of the shared library or the `mfilename` prototype name, if specified. *`COMPUTER`* is the string returned by the `computer` function. MATLAB compiles this file and creates the file:

```
BASENAME_thunk_COMPUTER.LIBEXT
```

where *`LIBEXT`* is the platform-dependent default shared library extension, for example, `dll` on Windows.

See Also

`mex` | `unloadlibrary` | `libisloaded` | `libfunctions` | `computer` | `calllib`

Related Examples

- "Create Alias Function Name Using Prototype File"

loadobj

Purpose

Modify load process for object

Syntax

```
b = loadobj(a)
```

Description

`b = loadobj(a)` is called by the `load` function if the class of `a` defines a `loadobj` method. `load` returns `b` as the value loaded from a MAT-file.

Define a `loadobj` method when objects of the class require special processing when loaded from MAT-files. If you define a `saveobj` method, then define a `loadobj` method to restore the object to the desired state. Define `loadobj` as a static method so it can accept as an argument whatever object or structure that you saved in the MAT-file. See "Implement loadobj as a Static Method".

When loading a subclass object, `load` calls only the subclass `loadobj` method. If a superclass defines a `loadobj` method, the subclass inherits this method. However, it is possible that the inherited method does not perform the necessary operations to load the subclass object. Consider overriding superclass `loadobj` methods.

If any superclass in a class hierarchy defines a `loadobj` method, then the subclass `loadobj` method must ensure that the subclass and superclass objects load properly. Ensure proper loading by calling the superclass `loadobj` (or other methods) from the subclass `loadobj` method. See "Saving and Loading Objects from Class Hierarchies".

Input Arguments

a

The input argument, `a`, can be:

- The object as loaded from the MAT-file.
- A structure created by `load` (if `load` cannot resolve the object).
- A `struct` returned by the `saveobj` method.

See Also

`load` | `save` | `saveobj`

Tutorials

- "Control Save and Load"

Purpose

Natural logarithm

Syntax

```
Y = log(X)
```

Description

The `log` function operates element-wise on arrays. Its domain includes complex and negative numbers, which may lead to unexpected results if used unintentionally.

`Y = log(X)` returns the natural logarithm of the elements of `X`. For complex or negative z where $z = x + y*i$, the complex logarithm is returned.

```
log(z) = log(abs(z)) + i*atan2(y,x)
```

Examples

The statement `abs(log(-1))` is a clever way to generate π.

```
ans =

    3.1416
```

See Also

`exp` | `log10` | `log2` | `logm` | `reallog`

log10

Purpose

Common (base 10) logarithm

Syntax

```
Y = log10(X)
```

Description

The `log10` function operates element-by-element on arrays. Its domain includes complex numbers, which may lead to unexpected results if used unintentionally.

`Y = log10(X)` returns the base 10 logarithm of the elements of `X`.

Examples

```
log10(realmax) is 308.2547
```

and

```
log10(eps) is -15.6536
```

See Also

`exp` | `log` | `log2` | `logm`

log1p

Purpose Compute log(1+x) accurately for small values of x

Syntax

```
y = log1p(x)
```

Description `y = log1p(x)` computes `log(1+x)`, compensating for the roundoff in `1+x`. `log1p(x)` is more accurate than `log(1+x)` for small values of x. For small `x`, `log1p(x)` is approximately `x`, whereas `log(1+x)` can be zero.

See Also `log` | `expm1`

log2

Purpose

Base 2 logarithm and dissect floating-point numbers into exponent and mantissa

Syntax

```
Y = log2(X)
[F,E] = log2(X)
```

Description

`Y = log2(X)` computes the base 2 logarithm of the elements of `X`.

`[F,E] = log2(X)` returns arrays `F` and `E`. Argument `F` is an array of real values, usually in the range `0.5 <= abs(F) < 1`. For real `X`, `F` satisfies the equation: `X = F.*2.^E`. Argument `E` is an array of integers that, for real `X`, satisfy the equation: `X = F.*2.^E`.

Tips

This function corresponds to the ANSI C function `frexp()` and the IEEE floating-point standard function `logb()`. Any zeros in `X` produce `F = 0` and `E = 0`.

Examples

For IEEE arithmetic, the statement `[F,E] = log2(X)` yields the values:

X	F	E
1	1/2	1
pi	pi/4	2
-3	-3/4	2
eps	1/2	-51
realmax	1-eps/2	1024
realmin	1/2	-1021

See Also

`log` | `pow2`

Purpose

Convert numeric values to logical

Syntax

```
K = logical(A)
```

Description

`K = logical(A)` returns an array that can be used for logical indexing or logical tests.

`A(B)`, where `B` is a logical array that is the same size as `A`, returns the values of `A` at the indices where the real part of `B` is nonzero.

`A(B)`, where `B` is a logical array that is smaller than `A`, returns the values of column vector `A(:)` at the indices where the real part of column vector `B(:)` is nonzero.

Tips

Most arithmetic operations remove the logicalness from an array. For example, adding zero to a logical array removes its logical characteristic. `A = +A` is the easiest way to convert a logical array, `A`, to a numeric double array.

Logical arrays are also created by the relational operators (==,<,>,~, etc.) and functions like `any`, `all`, `isnan`, `isinf`, and `isfinite`.

Examples

Given `A = [1 2 3; 4 5 6; 7 8 9]`, the statement `B = logical(eye(3))` returns a logical array

```
B =
   1    0    0
   0    1    0
   0    0    1
```

which can be used in logical indexing that returns `A`'s diagonal elements:

```
A(B)

ans =
     1
     5
     9
```

However, attempting to index into `A` using the *numeric* array `eye(3)` results in:

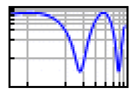
```
A(eye(3))
??? Subscript indices must either be real positive integers or
logicals.
```

See Also

`true` | `false` | `islogical` | Logical Operators: Elementwise | Logical Operators: Short-circuit

Purpose

Log-log scale plot



Syntax

```
loglog(Y)
loglog(X1,Y1,...)
loglog(X1,Y1,LineSpec,...)
loglog(...,'PropertyName',PropertyValue,...)
h = loglog(...)
```

Description

`loglog(Y)` plots the columns of `Y` versus their index if `Y` contains real numbers. If `Y` contains complex numbers, `loglog(Y)` and `loglog(real(Y),imag(Y))` are equivalent. `loglog` ignores the imaginary component in all other uses of this function.

`loglog(X1,Y1,...)` plots all `Xn` versus `Yn` pairs. If only `Xn` or `Yn` is a matrix, `loglog` plots the vector argument versus the rows or columns of the matrix, along the dimension of the matrix whose length matches the length of the vector. If the matrix is square, its columns plot against the vector if their lengths match.

`loglog(X1,Y1,`*`LineSpec`*`,...)` plots all lines defined by the `Xn,Yn,`*`LineSpec`* triples, where `LineSpec` determines line type, marker symbol, and color of the plotted lines. You can mix `Xn,Yn,`*`LineSpec`* triples with `Xn,Yn` pairs, for example,

```
loglog(X1,Y1,X2,Y2,LineSpec,X3,Y3)
```

`loglog(...,'`*`PropertyName`*`',PropertyValue,...)` sets property values for all lineseries properties graphics objects created by `loglog`.

`h = loglog(...)` returns a column vector of handles to lineseries graphics objects, one handle per line.

If you do not specify a color when plotting more than one line, `loglog` automatically cycles through the colors and line styles in the order specified by the current axes.

If you attempt to add a `loglog`, `semilogx`, or `semilogy` plot to a linear axis mode graph with `hold on`, the axis mode remains as it is and the new data plots as linear.

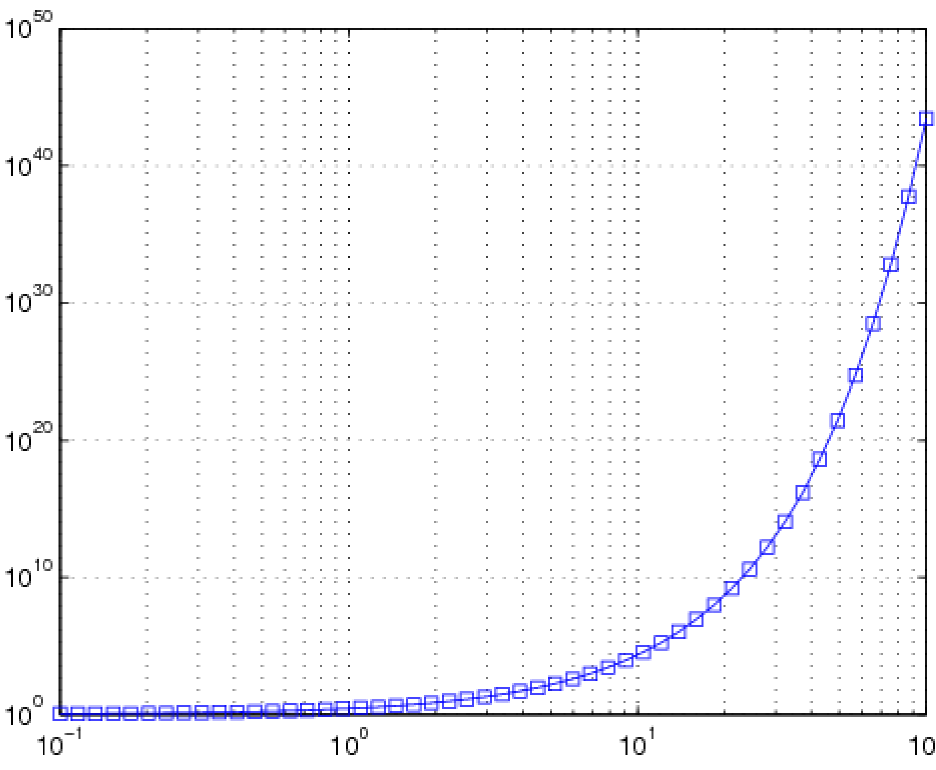
Renderer Support

The OpenGL renderer does not support logarithmic-scale axes. MATLAB automatically selects a different renderer when using logarithmic scaling. If you set the figure `Renderer` property to `opengl`, axis scales become linear. See the figure Renderer property for more information on renderers.

Examples

Create a simple `loglog` plot with square markers.

```
x = logspace(-1,2);
loglog(x,exp(x),'-s')
grid on
```



See Also

`LineSpec` | `plot` | `semilogx` | `semilogy`

Purpose Matrix logarithm

Syntax

```
L = logm(A)
[L, exitflag] = logm(A)
```

Description

`L = logm(A)` is the principal matrix logarithm of A, the inverse of `expm(A)`. `L` is the unique logarithm for which every eigenvalue has imaginary part lying strictly between $-\pi$ and π. If `A` is singular or has any eigenvalues on the negative real axis, the principal logarithm is undefined. In this case, `logm` computes a nonprincipal logarithm and returns a warning message.

`[L, exitflag] = logm(A)` returns a scalar `exitflag` that describes the exit condition of `logm`:

- If `exitflag = 0`, the algorithm was successfully completed.
- If `exitflag = 1`, too many matrix square roots had to be computed. However, the computed value of `L` might still be accurate.

The input `A` can have class `double` or `single`.

Tips

If `A` is real symmetric or complex Hermitian, then so is `logm(A)`.

Some matrices, like `A = [0 1; 0 0]`, do not have any logarithms, real or complex, so `logm` cannot be expected to produce one.

Limitations

For most matrices:

```
logm(expm(A)) = A = expm(logm(A))
```

These identities may fail for some `A`. For example, if the computed eigenvalues of `A` include an exact zero, then `logm(A)` generates infinity. Or, if the elements of `A` are too large, `expm(A)` may overflow.

Examples

Suppose `A` is the 3-by-3 matrix

```
1    1    0
0    0    2
```

```
     0          0         -1
```

and `Y = expm(A)` is

```
Y =
    2.7183    1.7183    1.0862
         0    1.0000    1.2642
         0         0    0.3679
```

Then `A = logm(Y)` produces the original matrix `A`.

```
Y =
    1.0000    1.0000    0.0000
         0         0    2.0000
         0         0   -1.0000
```

But `log(A)` involves taking the logarithm of zero, and so produces

```
ans =
    0.0000                   0            -35.5119
     -Inf                 -Inf              0.6931
     -Inf                 -Inf              0.0000 + 3.1416i
```

Algorithms

The algorithm `logm` uses is described in [1].

References

[1] Davies, P. I. and N. J. Higham, "A Schur-Parlett algorithm for computing matrix functions," *SIAM J. Matrix Anal. Appl.*, Vol. 25, Number 2, pp. 464-485, 2003.

[2] Cheng, S. H., N. J. Higham, C. S. Kenney, and A. J. Laub, "Approximating the logarithm of a matrix to specified accuracy," *SIAM J. Matrix Anal. Appl.*, Vol. 22, Number 4, pp. 1112-1125, 2001.

[3] Higham, N. J., "Evaluating Pade approximants of the matrix logarithm," *SIAM J. Matrix Anal. Appl.*, Vol. 22, Number 4, pp. 1126-1135, 2001.

[4] Golub, G. H. and C. F. Van Loan, *Matrix Computation*, Johns Hopkins University Press, 1983, p. 384.

[5] Moler, C. B. and C. F. Van Loan, “Nineteen Dubious Ways to Compute the Exponential of a Matrix,” *SIAM Review* 20, 1978, pp. 801-836.

See Also

`expm` | `funm` | `sqrtm`

Purpose

Generate logarithmically spaced vectors

Syntax

```
y = logspace(a,b)
y = logspace(a,b,n)
y = logspace(a,pi)
```

Description

The `logspace` function generates logarithmically spaced vectors. Especially useful for creating frequency vectors, it is a logarithmic equivalent of `linspace` and the ":" or colon operator.

`y = logspace(a,b)` generates a row vector `y` of 50 logarithmically spaced points between decades `10^a` and `10^b`.

`y = logspace(a,b,n)` generates `n` points between decades `10^a` and `10^b`.

`y = logspace(a,pi)` generates the points between `10^a` and `pi`, which is useful for digital signal processing where frequencies over this interval go around the unit circle.

Tips

All the arguments to `logspace` must be scalars.

See Also

`linspace` | `colon operator`

lookfor

Purpose

Search for keyword in all help entries

Alternatives

As an alternative to the `lookfor` function, use the Function Browser.

Syntax

```
lookfor topic
lookfor topic -all
```

Description

`lookfor topic` searches for the string `topic` in the first comment line (the H1 line) of the help text in all MATLAB program files found on the search path. For all files in which a match occurs, `lookfor` displays the H1 line.

`lookfor topic -all` searches the entire first comment block of a MATLAB program file looking for `topic`.

Examples

For example:

```
lookfor inverse
```

finds at least a dozen matches, including H1 lines containing "inverse hyperbolic cosine," "two-dimensional inverse FFT," and "pseudoinverse." Contrast this with

```
which inverse
```

or

```
what inverse
```

These functions run more quickly, but probably fail to find anything because MATLAB does not have a function `inverse`.

In summary, `what` lists the functions in a given folder, `which` finds the folder containing a given function or file, and `lookfor` finds all functions in all folders that might have something to do with a given keyword.

Even more extensive than the `lookfor` function are the find features in the Current Folder browser. For example, you can look for all occurrences of a specified word in all the MATLAB program files in the

current folder and its subfolders. For more information, see "Finding Files and Folders".

See Also

`dir` | `doc` | `filebrowser` | `strfind` | `help` | `regexp` | `what` | `which` | `who`

How To

- "Find Functions to Use"
- "Search Syntax and Tips"

lower

Purpose

Convert string to lowercase

Syntax

```
t = lower('str')
B = lower(A)
```

Description

`t = lower('str')` returns the string formed by converting any uppercase characters in `str` to the corresponding lowercase characters and leaving all other characters unchanged.

`B = lower(A)` when `A` is a cell array of strings, returns a cell array the same size as `A` containing the result of applying `lower` to each string within `A`.

Examples

`lower('MathWorks')` is `mathworks`.

Tips

Character sets supported:

- PC: Latin-1 for the Microsoft Windows operating system
- Other: ISO Latin-1 (ISO 8859-1)

See Also

`upper`

Purpose

List folder contents

Syntax

```
ls
ls name
list = ls(name)
```

Description

`ls` lists the contents of the current folder.

`ls name` lists the files and folders in the current folder that match the specified name. You can use wildcards.

`list = ls(name)` returns the files and folders in the current folder that match the specified `name` to `list`.

Tips

- On UNIX platforms, you can add any flags to `ls` that the operating system supports.

Input Arguments

name

A string value specifying a file or folder name.

Output Arguments

list

- On UNIX platforms, `list` is a character row vector of names separated by tab and space characters.
- On Microsoft Windows platforms, `list` is an `m`-by-`n` character array of names—`m` is the number of names and `n` is the number of characters in the longest name. MATLAB pads names shorter than `n` characters with space characters.

Examples

List all the files and folders in the current folder:

```
ls
```

List all the files and folders in the current folder that begin with the letter `h`:

```
ls h*
```

Return the list of all the files and folders in the current folder to `mylist`:

```
mylist = ls;
```

Alternatives

- View files and folders in the Current Folder browser.

  Open the Current Folder browser by issuing the `filebrowser` command.

See Also

`dir` | `pwd`

How To

- "Working with Files and Folders"

Purpose

Least-squares solution in presence of known covariance

Syntax

```
x = lscov(A,b)
x = lscov(A,b,w)
x = lscov(A,b,V)
x = lscov(A,b,V,alg)
[x,stdx] = lscov(...)
[x,stdx,mse] = lscov(...)
[x,stdx,mse,S] = lscov(...)
```

Description

`x = lscov(A,b)` returns the ordinary least squares solution to the linear system of equations `A*x = b`, i.e., `x` is the n-by-1 vector that minimizes the sum of squared errors `(b - A*x)'*(b - A*x)`, where `A` is m-by-n, and `b` is m-by-1. `b` can also be an m-by-k matrix, and `lscov` returns one solution for each column of `b`. When `rank(A) < n`, `lscov` sets the maximum possible number of elements of `x` to zero to obtain a "basic solution".

`x = lscov(A,b,w)`, where `w` is a vector length m of real positive weights, returns the weighted least squares solution to the linear system `A*x = b`, that is, `x` minimizes `(b - A*x)'*diag(w)*(b - A*x)`. `w` typically contains either counts or inverse variances.

`x = lscov(A,b,V)`, where `V` is an m-by-m real symmetric positive definite matrix, returns the generalized least squares solution to the linear system `A*x = b` with covariance matrix proportional to `V`, that is, `x` minimizes `(b - A*x)'*inv(V)*(b - A*x)`.

More generally, `V` can be positive semidefinite, and `lscov` returns `x` that minimizes `e'*e`, subject to `A*x + T*e = b`, where the minimization is over `x` and `e`, and `T*T' = V`. When `V` is semidefinite, this problem has a solution only if `b` is consistent with `A` and `V` (that is, `b` is in the column space of `[A T]`), otherwise `lscov` returns an error.

By default, `lscov` computes the Cholesky decomposition of `V` and, in effect, inverts that factor to transform the problem into ordinary least squares. However, if `lscov` determines that `V` is semidefinite, it uses an orthogonal decomposition algorithm that avoids inverting `V`.

`x = lscov(A,b,V,alg)` specifies the algorithm used to compute `x` when `V` is a matrix. `alg` can have the following values:

- `'chol'` uses the Cholesky decomposition of `V`.
- `'orth'` uses orthogonal decompositions, and is more appropriate when `V` is ill-conditioned or singular, but is computationally more expensive.

`[x,stdx] = lscov(...)` returns the estimated standard errors of `x`. When `A` is rank deficient, `stdx` contains zeros in the elements corresponding to the necessarily zero elements of `x`.

`[x,stdx,mse] = lscov(...)` returns the mean squared error. If `b` is assumed to have covariance matrix σ^2`V` (or (σ^2)×`diag`(1./W)), then `mse` is an estimate of σ^2.

`[x,stdx,mse,S] = lscov(...)` returns the estimated covariance matrix of `x`. When `A` is rank deficient, `S` contains zeros in the rows and columns corresponding to the necessarily zero elements of `x`. `lscov` cannot return `S` if it is called with multiple right-hand sides, that is, if `size(B,2) > 1`.

The standard formulas for these quantities, when `A` and `V` are full rank, are

- `x = inv(A'*inv(V)*A)*A'*inv(V)*B`
- `mse = B'*(inv(V) - inv(V)*A*inv(A'*inv(V)*A)*A'*inv(V))*B./(m-n)`
- `S = inv(A'*inv(V)*A)*mse`
- `stdx = sqrt(diag(S))`

However, `lscov` uses methods that are faster and more stable, and are applicable to rank deficient cases.

`lscov` assumes that the covariance matrix of `B` is known only up to a scale factor. `mse` is an estimate of that unknown scale factor, and `lscov` scales the outputs `S` and `stdx` appropriately. However, if `V` is known to be exactly the covariance matrix of `B`, then that scaling is unnecessary.

To get the appropriate estimates in this case, you should rescale S and stdx by 1/mse and sqrt(1/mse), respectively.

Algorithms

The vector x minimizes the quantity (A*x-b)'*inv(V)*(A*x-b). The classical linear algebra solution to this problem is

```
x = inv(A'*inv(V)*A)*A'*inv(V)*b
```

but the lscov function instead computes the QR decomposition of A and then modifies Q by V.

Examples

Example 1 — Computing Ordinary Least Squares

The MATLAB backslash operator (\) enables you to perform linear regression by computing ordinary least-squares (OLS) estimates of the regression coefficients. You can also use lscov to compute the same OLS estimates. By using lscov, you can also compute estimates of the standard errors for those coefficients, and an estimate of the standard deviation of the regression error term:

```
x1 = [.2 .5 .6 .8 1.0 1.1]';
x2 = [.1 .3 .4 .9 1.1 1.4]';
X = [ones(size(x1)) x1 x2];
y = [.17 .26 .28 .23 .27 .34]';

a = X\y
a =
    0.1203
    0.3284
   -0.1312

[b,se_b,mse] = lscov(X,y)
b =
    0.1203
    0.3284
   -0.1312
se_b =
    0.0643
```

```
    0.2267
    0.1488
mse =
    0.0015
```

Example 2 — Computing Weighted Least Squares

Use `lscov` to compute a weighted least-squares (WLS) fit by providing a vector of relative observation weights. For example, you might want to downweight the influence of an unreliable observation on the fit:

```
w = [1 1 1 1 1 .1]';

[bw,sew_b,msew] = lscov(X,y,w)
bw =
    0.1046
    0.4614
   -0.2621
sew_b =
    0.0309
    0.1152
    0.0814
msew =
  3.4741e-004
```

Example 3 — Computing General Least Squares

Use `lscov` to compute a general least-squares (GLS) fit by providing an observation covariance matrix. For example, your data may not be independent:

```
V = .2*ones(length(x1)) + .8*diag(ones(size(x1)));

[bg,sew_b,mseg] = lscov(X,y,V)
bg =
    0.1203
    0.3284
   -0.1312
sew_b =
```

```
    0.0672
    0.2267
    0.1488
mseg =
    0.0019
```

Example 4 — Estimating the Coefficient Covariance Matrix

Compute an estimate of the coefficient covariance matrix for either OLS, WLS, or GLS fits. The coefficient standard errors are equal to the square roots of the values on the diagonal of this covariance matrix:

```
[b,se_b,mse,S] = lscov(X,y);

S
S =
    0.0041   -0.0130    0.0075
   -0.0130    0.0514   -0.0328
    0.0075   -0.0328    0.0221

[se_b sqrt(diag(S))]
ans =
    0.0643    0.0643
    0.2267    0.2267
    0.1488    0.1488
```

References

[1] Strang, G., *Introduction to Applied Mathematics*, Wellesley-Cambridge, 1986, p. 398.

See Also

`lsqnonneg` | `qr` | `Arithmetic Operator \`

lsqnonneg

Purpose

Solve nonnegative least-squares constraints problem

Equation

Solves nonnegative least-squares curve fitting problems of the form

$$\min_x \|C \cdot x - d\|_2^2 \text{, where } x \geq 0.$$

Syntax

```
x = lsqnonneg(C,d)
x = lsqnonneg(C,d,options)
[x,resnorm] = lsqnonneg(...)
[x,resnorm,residual] = lsqnonneg(...)
[x,resnorm,residual,exitflag] = lsqnonneg(...)
[x,resnorm,residual,exitflag,output] = lsqnonneg(...)
[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg(...)
```

Description

`x = lsqnonneg(C,d)` returns the vector `x` that minimizes `norm(C*x-d)` subject to `x >= 0`. `C` and `d` must be real.

`x = lsqnonneg(C,d,options)` minimizes with the optimization parameters specified in the structure `options`. You can define these parameters using the `optimset` function. `lsqnonneg` uses these `options` structure fields:

`Display`	Level of display. `'off'` displays no output; `'final'` displays just the final output; `'notify'` (default) displays output only if the function does not converge.
`TolX`	Termination tolerance on `x`.

`[x,resnorm] = lsqnonneg(...)` returns the value of the squared 2-norm of the residual: `norm(C*x-d)^2`.

`[x,resnorm,residual] = lsqnonneg(...)` returns the residual, `d-C*x`.

`[x,resnorm,residual,exitflag] = lsqnonneg(...)` returns a value `exitflag` that describes the exit condition of `lsqnonneg`:

>0	Indicates that the function converged to a solution `x`.
0	Indicates that the iteration count was exceeded. Increasing the tolerance (`TolX` parameter in `options`) may lead to a solution.

`[x,resnorm,residual,exitflag,output] = lsqnonneg(...)` returns a structure `output` that contains information about the operation in the following fields:

`algorithm`	`'active-set'`
`iterations`	The number of iterations taken
`message`	Exit message

`[x,resnorm,residual,exitflag,output,lambda] = lsqnonneg(...)` returns the dual vector (Lagrange multipliers) `lambda`, where `lambda(i)<=0` when `x(i)` is (approximately) `0`, and `lambda(i)` is (approximately) `0` when `x(i)>0`.

Examples

Compare the unconstrained least squares solution to the `lsqnonneg` solution for a 4-by-2 problem:

```
C = [
    0.0372    0.2869
    0.6861    0.7071
    0.6233    0.6245
    0.6344    0.6170];
d = [
    0.8587
    0.1781
    0.0747
    0.8405];
[C\d lsqnonneg(C,d)] =
    -2.5627         0
     3.1108    0.6929
[norm(C*(C\d)-d) norm(C*lsqnonneg(C,d)-d)] =
     0.6674 0.9118
```

The solution from `lsqnonneg` does not fit as well (has a larger residual), as the least squares solution. However, the nonnegative least squares solution has no negative components.

Algorithms

`lsqnonneg` uses the algorithm described in [1]. The algorithm starts with a set of possible basis vectors and computes the associated dual vector `lambda`. It then selects the basis vector corresponding to the maximum value in `lambda` in order to swap out of the basis in exchange for another possible candidate. This continues until `lambda <= 0`.

References

[1] Lawson, C.L. and R.J. Hanson, *Solving Least Squares Problems*, Prentice-Hall, 1974, Chapter 23, p. 161.

See Also

`optimset` | `Arithmetic Operator \`

Purpose

LSQR method

Syntax

```
x = lsqr(A,b)
lsqr(A,b,tol)
lsqr(A,b,tol,maxit)
lsqr(A,b,tol,maxit,M)
lsqr(A,b,tol,maxit,M1,M2)
lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres,iter] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres,iter,resvec] = lsqr(A,b,tol,maxit,M1,M2,x0)
[x,flag,relres,iter,resvec,lsvec] = lsqr(A,b,tol,maxit,M1,M2,
   x0)
```

Description

`x = lsqr(A,b)` attempts to solve the system of linear equations `A*x=b` for `x` if `A` is consistent, otherwise it attempts to solve the least squares solution `x` that minimizes `norm(b-A*x)`. The `m`-by-`n` coefficient matrix `A` need not be square but it should be large and sparse. The column vector `b` must have length `m`. You can specify `A` as a function handle, `afun`, such that `afun(x,'notransp')` returns `A*x` and `afun(x,'transp')` returns `A'*x`.

"Parameterizing Functions" explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `lsqr` converges, a message to that effect is displayed. If `lsqr` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`lsqr(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `lsqr` uses the default, `1e-6`.

`lsqr(A,b,tol,maxit)` specifies the maximum number of iterations.

lsqr(A,b,tol,maxit,M) and lsqr(A,b,tol,maxit,M1,M2) use n-by-n preconditioner M or M = M1*M2 and effectively solve the system A*inv(M)*y = b for y, where y = M*x. If M is [] then lsqr applies no preconditioner. M can be a function mfun such that mfun(x,'notransp') returns M\x and mfun(x,'transp') returns M'\x.

lsqr(A,b,tol,maxit,M1,M2,x0) specifies the n-by-1 initial guess. If x0 is [], then lsqr uses the default, an all zero vector.

[x,flag] = lsqr(A,b,tol,maxit,M1,M2,x0) also returns a convergence flag.

Flag	Convergence
0	lsqr converged to the desired tolerance tol within maxit iterations.
1	lsqr iterated maxit times but did not converge.
2	Preconditioner M was ill-conditioned.
3	lsqr stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during lsqr became too small or too large to continue computing.

Whenever flag is not 0, the solution x returned is that with minimal norm residual computed over all the iterations. No messages are displayed if you specify the flag output.

[x,flag,relres] = lsqr(A,b,tol,maxit,M1,M2,x0) also returns an estimate of the relative residual norm(b-A*x)/norm(b). If flag is 0, relres <= tol.

[x,flag,relres,iter] = lsqr(A,b,tol,maxit,M1,M2,x0) also returns the iteration number at which x was computed, where 0 <= iter <= maxit.

[x,flag,relres,iter,resvec] = lsqr(A,b,tol,maxit,M1,M2,x0) also returns a vector of the residual norm estimates at each iteration, including norm(b-A*x0).

`[x,flag,relres,iter,resvec,lsvec] = lsqr(A,b,tol,maxit,M1,M2,x0)` also returns a vector of estimates of the scaled normal equations residual at each iteration: `norm((A*inv(M))'*(B-A*X))/norm(A*inv(M),'fro')`. Note that the estimate of `norm(A*inv(M),'fro')` changes, and hopefully improves, at each iteration.

Examples

Example 1

```
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);

x = lsqr(A,b,tol,maxit,M1,M2);
```

displays the following message:

```
lsqr converged at iteration 11 to a solution with relative
residual 3.5e-009
```

Example 2

This example replaces the matrix `A` in Example 1 with a handle to a matrix-vector product function `afun`. The example is contained in a function `run_lsqr` that

- Calls `lsqr` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_lsqr` are available to `afun`.

The following shows the code for `run_lsqr`:

```
function x1 = run_lsqr
n = 100;
```

```
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x1 = lsqr(@afun,b,tol,maxit,M1,M2);

    function y = afun(x,transp_flag)
       if strcmp(transp_flag,'transp')      % y = A'*x
          y = 4 * x;
          y(1:n-1) = y(1:n-1) - 2 * x(2:n);
          y(2:n) = y(2:n) - x(1:n-1);
       elseif strcmp(transp_flag,'notransp') % y = A*x
          y = 4 * x;
          y(2:n) = y(2:n) - 2 * x(1:n-1);
          y(1:n-1) = y(1:n-1) - x(2:n);
       end
    end
end
```

When you enter

```
x1=run_lsqr;
```

MATLAB software displays the message

```
lsqr converged at iteration 11 to a solution with relative
residual 3.5e-009
```

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Paige, C. C. and M. A. Saunders, "LSQR: An Algorithm for Sparse Linear Equations And Sparse Least Squares," *ACM Trans. Math. Soft.*, Vol.8, 1982, pp. 43-71.

See Also

bicg | bicgstab | cgs | gmres | minres | norm | pcg | qmr | symmlq | function_handle

Purpose

Test for less than

Syntax

```
A < B
lt(A, B)
```

Description

`A < B` compares each element of array `A` with the corresponding element of array `B`, and returns an array with elements set to logical 1 (`true`) where `A` is less than `B`, or set to logical 0 (`false`) where `A` is greater than or equal to `B`. Each input of the expression can be an array or a scalar value.

If both `A` and `B` are scalar (i.e., 1-by-1 matrices), then the MATLABsoftware returns a scalar value.

If both `A` and `B` are nonscalar arrays, then these arrays must have the same dimensions, and MATLAB returns an array of the same dimensions as `A` and `B`.

If one input is scalar and the other a nonscalar array, then the scalar input is treated as if it were an array having the same dimensions as the nonscalar input array. In other words, if input `A` is the number 100, and `B` is a 3-by-5 matrix, then `A` is treated as if it were a 3-by-5 matrix of elements, each set to 100. MATLAB returns an array of the same dimensions as the nonscalar input array.

`lt(A, B)` is called for the syntax `A < B` when either `A` or `B` is an object.

Examples

Create two 6-by-6 matrices, `A` and `B`, and locate those elements of `A` that are less than the corresponding elements of `B`:

```
A = magic(6);
B = repmat(3*magic(3), 2, 2);

A < B
ans =
     0     1     1     0     0     0
     1     0     1     0     0     0
     0     1     1     0     0     0
     1     0     0     1     0     1
```

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

See Also

`gt` | `le` | `ge` | `ne` | `eq`

How To

- "Relational Operators"

Purpose

LU matrix factorization

Syntax

```
Y = lu(A)
[L,U] = lu(A)
[L,U,P] = lu(A)
[L,U,P,Q] = lu(A)
[L,U,P,Q,R] = lu(A)
[...] = lu(A,'vector')
[...] = lu(A,thresh)
[...] = lu(A,thresh,'vector')
```

Description

The `lu` function expresses a matrix `A` as the product of two essentially triangular matrices, one of them a permutation of a lower triangular matrix and the other an upper triangular matrix. The factorization is often called the *LU*, or sometimes the *LR*, factorization. `A` can be rectangular.

`Y = lu(A)` returns matrix `Y` that, for sparse `A`, contains the strictly lower triangular `L`, i.e., without its unit diagonal, and the upper triangular `U` as submatrices. That is, if `[L,U,P] = lu(A)`, then `Y = U+L-eye(size(A))`. For nonsparse `A`, `Y` is the output from the LAPACK `dgetrf` or `zgetrf` routine. The permutation matrix `P` is not returned.

`[L,U] = lu(A)` returns an upper triangular matrix in `U` and a permuted lower triangular matrix in `L` such that `A = L*U`. Return value `L` is a product of lower triangular and permutation matrices.

`[L,U,P] = lu(A)` returns an upper triangular matrix in `U`, a lower triangular matrix `L` with a unit diagonal, and a permutation matrix `P`, such that `L*U = P*A`. The statement `lu(A,'matrix')` returns identical output values.

`[L,U,P,Q] = lu(A)` for sparse nonempty `A`, returns a unit lower triangular matrix `L`, an upper triangular matrix `U`, a row permutation matrix `P`, and a column reordering matrix `Q`, so that `P*A*Q = L*U`. If `A` is empty or not sparse, `lu` displays an error message. The statement `lu(A,'matrix')` returns identical output values.

`[L,U,P,Q,R] = lu(A)` returns unit lower triangular matrix `L`, upper triangular matrix `U`, permutation matrices `P` and `Q`, and a diagonal scaling matrix `R` so that `P*(R\A)*Q = L*U` for sparse non-empty `A`. Typically, but not always, the row-scaling leads to a sparser and more stable factorization. The statement `lu(A,'matrix')` returns identical output values.

`[...] = lu(A,'vector')` returns the permutation information in two row vectors `p` and `q`. You can specify from 1 to 5 outputs. Output `p` is defined as `A(p,:)=L*U`, output `q` is defined as `A(p,q)=L*U`, and output `R` is defined as `R(:,p)\A(:,q)=L*U`.

`[...] = lu(A,thresh)` controls pivoting. This syntax applies to sparse matrices only. The `thresh` input is a one- or two-element vector of type `single` or `double` that defaults to [0.1, 0.001]. If `A` is a square matrix with a mostly symmetric structure and mostly nonzero diagonal, MATLAB uses a symmetric pivoting strategy. For this strategy, the diagonal where

```
A(i,j) >= thresh(2) * max(abs(A(j:m,j)))
```

is selected. If the diagonal entry fails this test, a pivot entry below the diagonal is selected, using `thresh(1)`. In this case, `L` has entries with absolute value `1/min(thresh)` or less.

If A is not as described above, MATLAB uses an asymmetric strategy. In this case, the sparsest row `i` where

```
A(i,j) >= thresh(1) * max(abs(A(j:m,j)))
```

is selected. A value of 1.0 results in conventional partial pivoting. Entries in `L` have an absolute value of `1/thresh(1)` or less. The second element of the `thresh` input vector is not used when MATLAB uses an asymmetric strategy.

Smaller values of `thresh(1)` and `thresh(2)` tend to lead to sparser LU factors, but the solution can become inaccurate. Larger values can lead to a more accurate solution (but not always), and usually an increase in the total work and memory usage. The statement `lu(A,thresh,'matrix')` returns identical output values.

`[...] = lu(A,thresh,'vector')` controls the pivoting strategy and also returns the permutation information in row vectors, as described above. The `thresh` input must precede `'vector'` in the input argument list.

Note In rare instances, incorrect factorization results in `P*A*Q` ≠ `L*U`. Increase `thresh`, to a maximum of `1.0` (regular partial pivoting), and try again.

Tips

Most of the algorithms for computing LU factorization are variants of Gaussian elimination. The factorization is a key step in obtaining the inverse with `inv` and the determinant with `det`. It is also the basis for the linear equation solution or matrix division obtained with `\` and `/`.

Arguments

`A`	Rectangular matrix to be factored.
`thresh`	Pivot threshold for sparse matrices. Valid values are in the interval `[0,1]`. If you specify the fourth output `Q`, the default is `0.1`. Otherwise, the default is `1.0`.
`L`	Factor of `A`. Depending on the form of the function, `L` is either a unit lower triangular matrix, or else the product of a unit lower triangular matrix with `P'`.
`U`	Upper triangular matrix that is a factor of `A`.
`P`	Row permutation matrix satisfying the equation `L*U = P*A`, or `L*U = P*A*Q`. Used for numerical stability.
`Q`	Column permutation matrix satisfying the equation `P*A*Q = L*U`. Used to reduce fill-in in the sparse case.
`R`	Row-scaling matrix

Examples

Example 1

Start with

```
A = [ 1    2    3
      4    5    6
      7    8    0 ];
```

To see the LU factorization, call `lu` with two output arguments.

```
[L1,U] = lu(A)

L1 =
    0.1429    1.0000         0
    0.5714    0.5000    1.0000
    1.0000         0         0

U =
    7.0000    8.0000         0
         0    0.8571    3.0000
         0         0    4.5000
```

Notice that `L1` is a permutation of a lower triangular matrix: if you switch rows 2 and 3, and then switch rows 1 and 2, the resulting matrix is lower triangular and has `1`s on the diagonal. Notice also that `U` is upper triangular. To check that the factorization does its job, compute the product

```
L1*U
```

which returns the original `A`. The inverse of the example matrix, `X = inv(A)`, is actually computed from the inverses of the triangular factors

```
X = inv(U)*inv(L1)
```

Using three arguments on the left side to get the permutation matrix as well,

```
[L2,U,P] = lu(A)
```

returns a truly lower triangular L2, the same value of U, and the permutation matrix P.

```
L2 =

    1.0000         0         0
    0.1429    1.0000         0
    0.5714    0.5000    1.0000

U =
    7.0000    8.0000         0
         0    0.8571    3.0000
         0         0    4.5000

P =
    0    0    1
    1    0    0
    0    1    0
```

Note that L2 = P*L1.

```
P*L1

ans =

    1.0000         0         0
    0.1429    1.0000         0
    0.5714    0.5000    1.0000
```

To verify that L2*U is a permuted version of A, compute L2*U and subtract it from P*A:

```
P*A - L2*U

ans =
     0     0     0
     0     0     0
     0     0     0
```

In this case, `inv(U)*inv(L)` results in the permutation of `inv(A)` given by `inv(P)*inv(A)`.

The determinant of the example matrix is

```
d = det(A)

d =    27
```

It is computed from the determinants of the triangular factors

```
d = det(L)*det(U)
```

The solution to $Ax = b$ is obtained with matrix division

```
x = A\b
```

The solution is actually computed by solving two triangular systems

```
y = L\b
x = U\y
```

Example 2

The 1-norm of their difference is within roundoff error, indicating that `L*U = P*B*Q`.

Generate a 60-by-60 sparse adjacency matrix of the connectivity graph of the Buckminster-Fuller geodesic dome.

```
B = bucky;
```

Use the sparse matrix syntax with four outputs to get the row and column permutation matrices.

```
[L,U,P,Q] = lu(B);
```

Apply the permutation matrices to `B`, and subtract the product of the lower and upper triangular matrices.

```
Z = P*B*Q - L*U;
```

```
norm(Z,1)

ans =
  7.9936e-015
```

Example 3

This example illustrates the benefits of using the 'vector' option. Note how much memory is saved by using the lu(F,'vector') syntax.

```
F = gallery('uniformdata',[1000 1000],0);
g = sum(F,2);
[L,U,P] = lu(F);
[L,U,p] = lu(F,'vector');
whos P p

Name            Size            Bytes  Class     Attributes
P          1000x1000          8000000  double
p             1x1000             8000  double
```

The following two statements are equivalent. The first typically requires less time:

```
x = U\(L\(g(p,:)));
y = U\(L\(P*g));
```

See Also

cond | det | inv | luinc | qr | rref | Arithmetic Operators \,/

Purpose

Sparse incomplete LU factorization

Note `luinc` has been removed. Use `ilu` instead.

Syntax

```
luinc(A,'0')
luinc(A,droptol)
luinc(A,options)
[L,U] = luinc(A,'0')
[L,U] = luinc(A,options)
[L,U,P] = luinc(...)
```

Description

`luinc` produces a unit lower triangular matrix, an upper triangular matrix, and a permutation matrix.

`luinc(A,'0')` computes the incomplete LU factorization of level `0` of a square sparse matrix. The triangular factors have the same sparsity pattern as the permutation of the original sparse matrix `A`, and their product agrees with the permuted `A` over its sparsity pattern. `luinc(A,'0')` returns the strict lower triangular part of the factor and the upper triangular factor embedded within the same matrix. The permutation information is lost, but `nnz(luinc(A,'0')) = nnz(A)`, with the possible exception of some zeros due to cancellation.

`luinc(A,droptol)` computes the incomplete LU factorization of any sparse matrix using the drop tolerance specified by the non-negative scalar `droptol`. The result is an approximation of the complete LU factors returned by `lu(A)`. For increasingly smaller values of the drop tolerance, this approximation improves until the drop tolerance is `0`, at which time the complete LU factorization is produced, as in `lu(A)`.

As each column `j` of the triangular incomplete factors is being computed, the entries smaller in magnitude than the local drop tolerance (the product of the drop tolerance and the norm of the corresponding column of `A`)

```
droptol*norm(A(:,j))
```

are dropped from the appropriate factor.

The only exceptions to this dropping rule are the diagonal entries of the upper triangular factor, which are preserved to avoid a singular factor.

`luinc(A,options)` computes the factorization with up to four options. These options are specified by fields of the input structure `options`. The fields must be named exactly as shown in the table below. You can include any number of these fields in the structure and define them in any order. Any additional fields are ignored.

Field Name	Description
`droptol`	Drop tolerance of the incomplete factorization.
`milu`	If `milu` is `1`, `luinc` produces the modified incomplete LU factorization that subtracts the dropped elements in any column from the diagonal element of the upper triangular factor. The default value is `0`.
`udiag`	If `udiag` is `1`, any zeros on the diagonal of the upper triangular factor are replaced by the local drop tolerance. The default is `0`.
`thresh`	Pivot threshold between `0` (forces diagonal pivoting) and `1`, the default, which always chooses the maximum magnitude entry in the column to be the pivot. `thresh` is described in greater detail in the `lu` reference page.

`luinc(A,options)` is the same as `luinc(A,droptol)` if options has `droptol` as its only field.

`[L,U] = luinc(A,'0')` returns the product of permutation matrices and a unit lower triangular matrix in `L` and an upper triangular matrix in `U`. The exact sparsity patterns of `L`, `U`, and `A` are not comparable but the number of nonzeros is maintained with the possible exception of some zeros in `L` and `U` due to cancellation:

```
nnz(L)+nnz(U) = nnz(A)+n, where A is n-by-n.
```

The product L*U agrees with A over its sparsity pattern. (L*U).*spones(A)-A has entries of the order of eps.

[L,U] = luinc(A,options) returns a permutation of a unit lower triangular matrix in L and an upper triangular matrix in U. The product L*U is an approximation to A. luinc(A,options) returns the strict lower triangular part of the factor and the upper triangular factor embedded within the same matrix. The permutation information is lost.

[L,U,P] = luinc(...) returns a unit lower triangular matrix in L, an upper triangular matrix in U, and a permutation matrix in P.

[L,U,P] = luinc(A,'0') returns a unit lower triangular matrix in L, an upper triangular matrix in U and a permutation matrix in P. L has the same sparsity pattern as the lower triangle of permuted A

```
spones(L) = spones(tril(P*A))
```

with the possible exceptions of 1s on the diagonal of L where P*A may be zero, and zeros in L due to cancellation where P*A may be nonzero. U has the same sparsity pattern as the upper triangle of P*A

```
spones(U) = spones(triu(P*A))
```

with the possible exceptions of zeros in U due to cancellation where P*A may be nonzero. The product L*U agrees within rounding error with the permuted matrix P*A over its sparsity pattern. (L*U).*spones(P*A)-P*A has entries of the order of eps.

[L,U,P] = luinc(A,options) returns a unit lower triangular matrix in L, an upper triangular matrix in U, and a permutation matrix in P. The nonzero entries of U satisfy

```
abs(U(i,j)) >= droptol*norm((A:,j)),
```

with the possible exception of the diagonal entries, which were retained despite not satisfying the criterion. The entries of L were tested against the local drop tolerance before being scaled by the pivot, so for nonzeros in L

```
abs(L(i,j)) >= droptol*norm(A(:,j))/U(j,j).
```

The product `L*U` is an approximation to the permuted `P*A`.

Tips

These incomplete factorizations may be useful as preconditioners for solving large sparse systems of linear equations. The lower triangular factors all have `1`s along the main diagonal but a single `0` on the diagonal of the upper triangular factor makes it singular. The incomplete factorization with a drop tolerance prints a warning message if the upper triangular factor has zeros on the diagonal. Similarly, using the `udiag` option to replace a zero diagonal only gets rid of the symptoms of the problem but does not solve it. The preconditioner may not be singular, but it probably is not useful and a warning message is printed.

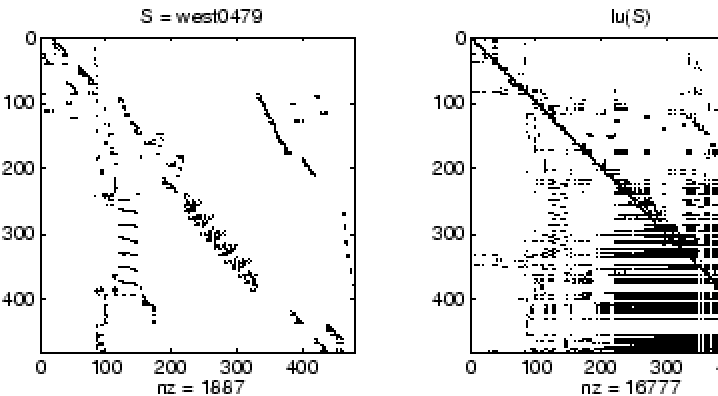
Limitations

`luinc(X,'0')` works on square matrices only.

Examples

Start with a sparse matrix and compute its LU factorization.

```
load west0479;
S = west0479;
[L,U] = lu(S);
```



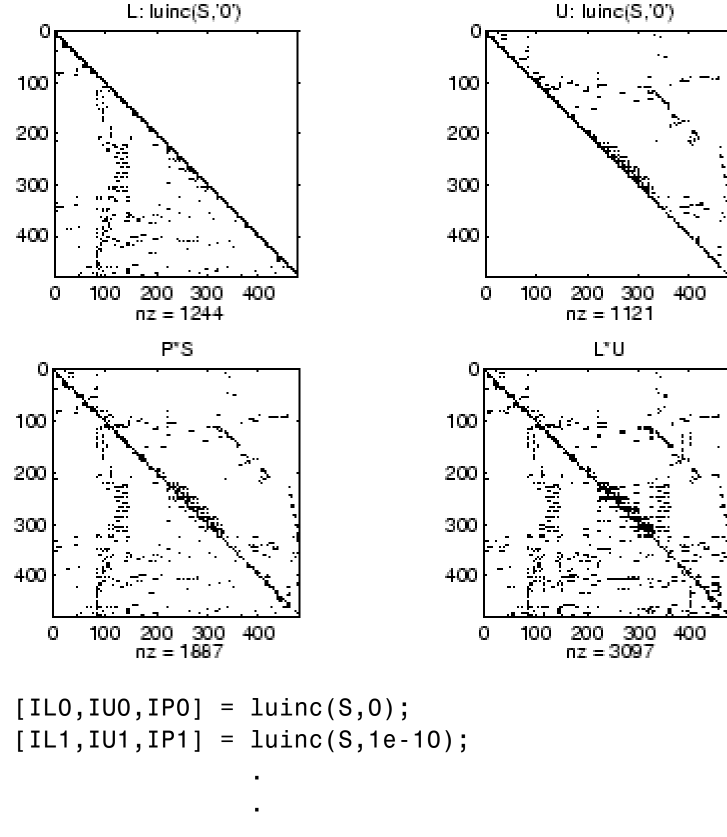
Compute the incomplete LU factorization of level 0.

```
[L,U,P] = luinc(S,'0');
```

```
D = (L*U).*spones(P*S)-P*S;
```

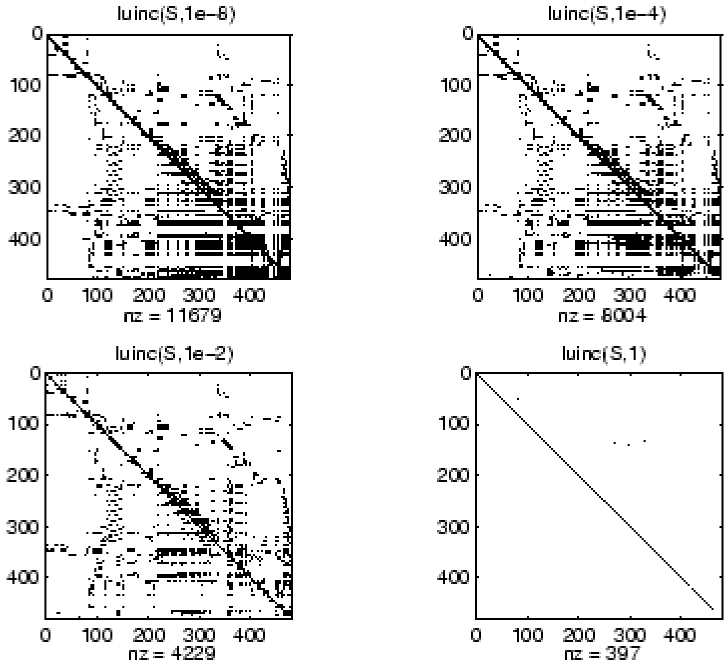
spones(U) and spones(triu(P*S)) are identical.

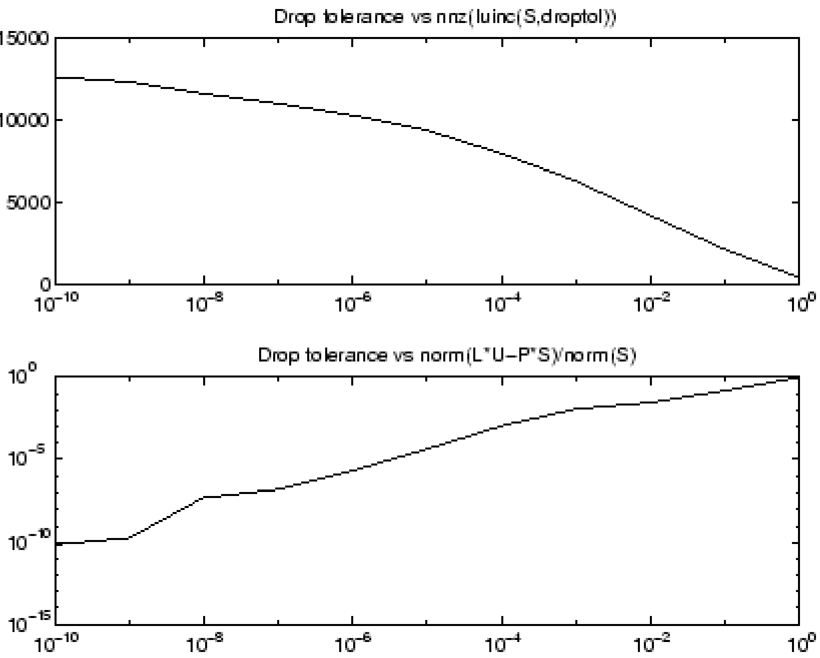
spones(L) and spones(tril(P*S)) disagree at 73 places on the diagonal, where L is 1 and P*S is 0, and also at position (206,113), where L is 0 due to cancellation, and P*S is -1. D has entries of the order of eps.




```
[IL0,IU0,IP0] = luinc(S,0);
[IL1,IU1,IP1] = luinc(S,1e-10);
                .
                .
                .
```

A drop tolerance of `0` produces the complete LU factorization. Increasing the drop tolerance increases the sparsity of the factors (decreases the number of nonzeros) but also increases the error in the factors, as seen in the plot of drop tolerance versus `norm(L*U-P*S,1)/norm(S,1)` in the second figure below.





Algorithms

`luinc(A,'0')` is based on the "KJI" variant of the LU factorization with partial pivoting. Updates are made only to positions which are nonzero in `A`.

`luinc(A,droptol)` and `luinc(A,options)` are based on the column-oriented `lu` for sparse matrices.

References

[1] Saad, Yousef, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996, Chapter 10 - Preconditioning Techniques.

See Also

`bicg` | `cholinc` | `ilu` | `lu`

Purpose

Magic square

Syntax

```
M = magic(n)
```

Description

`M = magic(n)` returns an `n`-by-`n` matrix constructed from the integers `1` through `n^2` with equal row and column sums. The order n must be a scalar greater than or equal to `3`.

Tips

A magic square, scaled by its magic sum, is doubly stochastic.

Examples

The magic square of order 3 is

```
M = magic(3)

M =

    8    1    6
    3    5    7
    4    9    2
```

This is called a magic square because the sum of the elements in each column is the same.

```
 sum(M) =

     15     15     15
```

And the sum of the elements in each row, obtained by transposing twice, is the same.

```
 sum(M')' =

     15
     15
     15
```

This is also a special magic square because the diagonal elements have the same sum.

```
sum(diag(M)) =

    15
```

The value of the characteristic sum for a magic square of order `n` is

```
sum(1:n^2)/n
```

which, when `n = 3`, is `15`.

Algorithms

There are three different algorithms:

- `n` odd
- `n` even but not divisible by four
- `n` divisible by four

To make this apparent, type

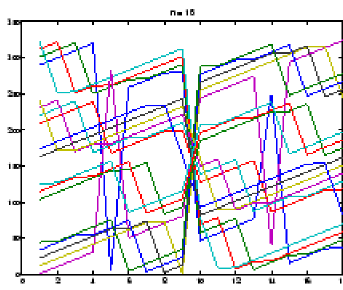
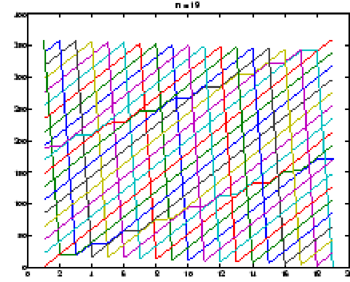
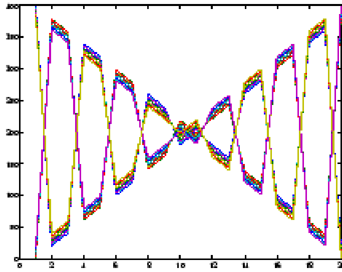
```
for n = 3:20
    A = magic(n);
    r(n) = rank(A);
end
```

For `n` odd, the rank of the magic square is `n`. For `n` divisible by 4, the rank is `3`. For `n` even but not divisible by 4, the rank is `n/2 + 2`.

```
[(3:20)',r(3:20)']
ans =
     3     3
     4     3
     5     5
     6     5
     7     7
     8     3
     9     9
    10     7
    11    11
    12     3
```

```
13      13
14       9
15      15
16       3
17      17
18      11
19      19
20       3
```

Plotting A for n = 18, 19, 20 shows the characteristic plot for each category.

Limitations

If you supply `n` less than `3`, `magic` returns either a nonmagic square, or else the degenerate magic squares `1` and `[]`.

See Also

`ones` | `rand`

Purpose

Create 4-by-4 transform matrix

Syntax

```
M = makehgtform
M = makehgtform('translate',[tx ty tz])
M = makehgtform('scale',s)
M = makehgtform('scale',[sx,sy,sz])
M = makehgtform('xrotate',t)
M = makehgtform('yrotate',t)
M = makehgtform('zrotate',t)
M = makehgtform('axisrotate',[ax,ay,az],t)
```

Description

Use `makehgtform` to create transform matrices for translation, scaling, and rotation of graphics objects. Apply the transform to graphics objects by assigning the transform to the `Matrix` property of a parent hgtransform object. See Examples for more information.

`M = makehgtform` returns an identity transform.

`M = makehgtform('translate',[tx ty tz])` or M = makehgtform('translate',tx,ty,tz) returns a transform that translates along the *x*-axis by `tx`, along the *y*-axis by `ty`, and along the *z*-axis by `tz`.

`M = makehgtform('scale',s)` returns a transform that scales uniformly along the *x*-, *y*-, and *z*-axes.

`M = makehgtform('scale',[sx,sy,sz])` returns a transform that scales along the *x*-axis by `sx`, along the *y*-axis by `sy`, and along the *z*-axis by `sz`.

`M = makehgtform('xrotate',t)` returns a transform that rotates around the *x*-axis by `t` radians.

`M = makehgtform('yrotate',t)` returns a transform that rotates around the *y*-axis by `t` radians.

`M = makehgtform('zrotate',t)` returns a transform that rotates around the *z*-axis by `t` radians.

`M = makehgtform('axisrotate',[ax,ay,az],t)` Rotate around axis `[ax ay az]` by `t` radians.

Note that you can specify multiple operations in one call to `makehgtform` and the MATLAB software returns a transform matrix that is the result of concatenating all specified operations. For example,

```
m = makehgtform('xrotate',pi/2,'yrotate',pi/2);
```

is the same as

```
m = makehgtform('xrotate',pi/2)*makehgtform('yrotate',pi/2);
```

See Also

`hggroup` | `hgtransform` | `Hgtransform Properties`

How To

- "Group Objects"

Purpose

Map values to unique keys

Description

A Map object is a data structure that allows you to retrieve values using a corresponding key. Keys can be real numbers or text strings and provide more flexibility for data access than array indices, which must be positive integers. Values can be scalar or nonscalar arrays.

Construction

`mapObj = containers.Map` constructs an empty Map container `mapObj`.

`mapObj = containers.Map(keySet,valueSet)` constructs a Map that contains one or more values and a unique key for each value.

`mapObj = containers.Map(keySet,valueSet,'UniformValues',isUniform)` specifies whether all values must be uniform (either all scalars of the same data type, or all strings). Possible values for `isUniform` are logical `true` (1) or `false` (0).

`mapObj = containers.Map('KeyType',kType,'ValueType',vType)` constructs an empty Map object and sets the KeyType and ValueType properties. The order of the key type and value type argument pairs is not important, but both pairs are required.

Input Arguments

keySet

1-by-n array that specifies n unique keys for the map.

All keys in a Map object are real numeric values or all keys are strings. If n > 1 and the keys are strings, `keySet` must be a cell array. The number of keys in `keySet` must equal the number of values in `valueSet`.

valueSet

1-by-n array of any class that specifies n values for the map. The number of values in `valueSet` must equal the number of keys in `keySet`.

'UniformValues'

Parameter string to use with the `isUniform` argument.

isUniform

Logical value that specifies whether all values are uniform. If `isUniform` is `true` (1), all values must be scalars of the same data type, or all values must be strings. If `isUniform` is `false` (0), then `containers.Map` sets the `ValueType` to `'any'`.

Default: `true` for empty `Map` objects, otherwise determined by the data types of values in `valueSet`.

'KeyType'

Parameter string to use with the `kType` argument.

kType

String that specifies the data type for the keys. Possible values are `'char'`, `'double'`, `'single'`, `'int32'`, `'uint32'`, `'int64'`, or `'uint64'`.

Default: `'char'` for empty `Map` objects, otherwise determined by the data types of keys in `keySet`. If you specify keys of different numeric types, `kType` is `'double'`.

'ValueType'

Literal string parameter to use with the `vType` argument.

vType

String that specifies the data type for the keys. Possible values are `'any'`, `'char'`, `'logical'`, `'double'`, `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, or `'uint64'`.

Default: `'any'` when you create an empty `Map` object or when you specify values of different sizes or types, otherwise determined by the data type of `valueSet`.

Properties

Count

Unsigned 64-bit integer that represents the total number of key-value pairs contained in the Map object. Read only.

KeyType

Character array that indicates the data type of all keys in the Map object. The default KeyType for empty Map objects is 'char'. Otherwise, KeyType is determined from the data type of the keySet inputs. Read only.

ValueType

Character array that indicates the data type of all values in the Map object. If you construct an empty Map object or specify values with different data types, then the value of ValueType is 'any'. Otherwise, ValueType is determined from the data type of the valueSet inputs. Read only.

Methods

isKey	Determine if containers.Map object contains key
keys	Identify keys of containers.Map object
length	Length of containers.Map object
remove	Remove key-value pairs from containers.Map object
size	Size of containers.Map object
values	Identify values in containers.Map object

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Construct a Map and View Properties

Construct a Map object that contains rainfall data for several months:

```
keySet =   {'Jan', 'Feb', 'Mar', 'Apr'};
valueSet = [327.2, 368.2, 197.6, 178.4];
mapObj = containers.Map(keySet,valueSet)
```

This code returns a description of the map, including the property values:

```
mapObj =

  containers.Map handle
  Package: containers

  Properties:
        Count: 4
      KeyType: 'char'
    ValueType: 'double'

  Methods, Events, Superclasses
```

Get a specific property using dot notation, such as

```
mapObj.Count
```

which returns

```
ans =
                    4
```

Look Up Values in a Map

Use the map created in the previous example to find the rainfall data for February:

```
rainFeb = mapObj('Feb')
```

This code returns

```
rainFeb =
  368.2000
```

Add a Single Value and Key to a Map

Add data for the month of May to the map created in the first example:

```
mapObj('May') = 100.0;
```

Add Multiple Values and Keys by Concatenating Maps

Create a map that contains rainfall data for June, July, and August, and add the data to `mapObj` (from previous examples):

```
keySet   = {'Jun','Jul','Aug'};
valueSet = [ 69.9, 32.3, 37.3];
newMap = containers.Map(keySet,valueSet);

mapObj = [mapObj; newMap];
```

Map objects only support vertical concatenation (that is, adding columns with a semicolon, `;`). When concatenating maps, the data type of all values must be consistent with the `ValueType` of the leftmost map. In this example, both maps have the a `ValueType` of `double`.

Get the Keys or Values in a Map

Determine all the keys of `mapObj` (from previous examples) by calling the `keys` method:

```
allKeys = keys(mapObj)
```

This method returns the keys in alphabetical order:

```
allKeys =
    'Apr'    'Aug'    'Feb'    'Jan'    'Jul'    'Jun'    'Mar'    'Ma
```

Get multiple values from the map by calling the `values` method. Like the `keys` method, you can request all values with the syntax `values(mapObj)`. Alternatively, request values for specific keys. For example, view the values for March, April, and May in `mapObj`:

```
springValues = values(mapObj,{'Mar','Apr','May'})
```

This method returns the values in a cell array, in the order corresponding to the specified keys:

```
springValues =
    [197.6000]    [178.4000]    [100]
```

Remove Keys and Values

Remove the data for March and April from `mapObj` (from previous examples) by calling the `remove` method, and view the remaining keys:

```
remove(mapObj,{'Mar','Apr'});
keys(mapObj)
```

This code returns

```
ans =
    'Aug'    'Feb'    'Jan'    'Jul'    'Jun'    'May'
```

Create a Map with Nonscalar Values

Map integer keys to nonscalar arrays, and view the value for one of the keys:

```
keySet = [5,10,15];
valueSet = {magic(5),magic(10),magic(15)};
mapObj = containers.Map(keySet,valueSet);
mapObj(5)
```

This code returns

```
ans =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

Construct an Empty Map

Construct a map with no values, but set the `KeyType` and `ValueType` properties:

```
mapObj = containers.Map('KeyType','char','ValueType','int32')
```

This code returns

```
mapObj =

  containers.Map handle
  Package: containers

  Properties:
        Count: 0
      KeyType: 'char'
    ValueType: 'int32'

  Methods, Events, Superclasses
```

Specify Whether Values Are Uniform

Construct a map with numeric values, and specify that the values do not have to be uniform:

```
keySet = {'a','b','c'};
valueSet = {1,2,3};
mapObj = containers.Map(keySet,valueSet,'UniformValues',false);
```

This map allows nonnumeric values, so

```
mapObj('d') = 'OK';
values(mapObj)
```

returns

```
ans =
    [1]    [2]    [3]    'OK'
```

See Also

keys | isKey | values

Purpose

Convert array to cell array with potentially different sized cells

Syntax

```
C = mat2cell(A,dim1Dist,...,dimNDist)
C = mat2cell(A,rowDist)
```

Description

`C = mat2cell(A,dim1Dist,...,dimNDist)` divides array `A` into smaller arrays within cell array `C`. Vectors `dim1Dist,...dimNDist` specify how to divide the rows, columns, and (when applicable) higher dimensions of `A`.

`C = mat2cell(A,rowDist)` divides array `A` into an `n`-by-1 cell array `C`, where `n == numel(rowDist)`.

Input Arguments

A

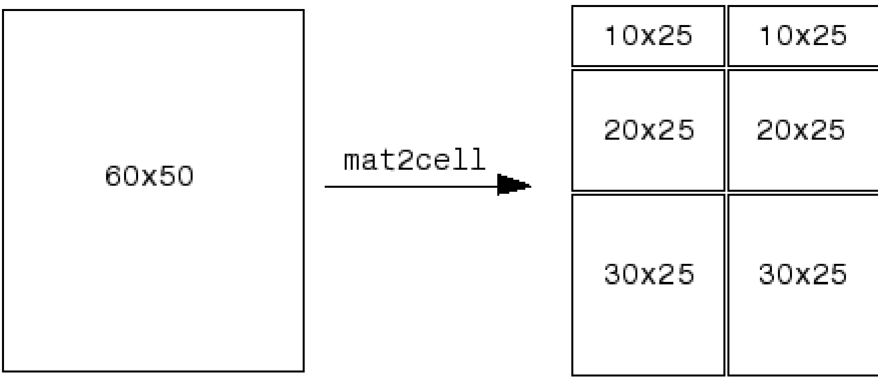
Any type of array.

dim1Dist,...,dimNDist

Numeric vectors that describe how to divide each dimension of `A`. For example, this command

```
c = mat2cell(x, [10, 20, 30], [25, 25])
```

divides a 60-by-50 array into six arrays contained in a cell array.



For the *k*th dimension, `sum(dimkDist) == size(A, k)`.

If the *k*th dimension of `A` is zero, set the corresponding `dimkDist` to the empty array, `[]`. For example,

```
a = rand(3, 0, 4);
c = mat2cell(a, [1, 2], [], [2, 1, 1]);
```

rowDist

Numeric vector that describes how to divide the rows of `A`. When you do not specify distributions for any other dimension, the `mat2cell` function creates an `n`-by-1 cell array `C`, where `n == numel(rowDist)`.

Output Arguments

C

Cell array. The *k*th dimension of array `C` is given by `size(C, k) == numel(dimkDist)`. The *k*th dimension of the *i*th cell of `C` is given by `size(C{i}, k) == dimkDist(i)`.

Examples

Divide the 5-by-4 matrix `X` into 2-by-3 and 2-by-2 matrices contained in a cell array.

```
X = reshape(1:20,5,4)'
C = mat2cell(X, [2 2], [3 2])
celldisp(C)
```

This code returns

```
X =
     1     2     3     4     5
     6     7     8     9    10
    11    12    13    14    15
    16    17    18    19    20

C =
    [2x3 double]    [2x2 double]
    [2x3 double]    [2x2 double]
```

```
C{1,1} =
     1     2     3
     6     7     8

C{2,1} =
    11    12    13
    16    17    18

C{1,2} =
     4     5
     9    10

C{2,2} =
    14    15
    19    20
```

Divide X (created in the previous example) into a 2-by-1 cell array.

```
C = mat2cell(X, [1 3])
celldisp(C)
```

This code returns

```
C =
    [1x5 double]
    [3x5 double]

C{1} =
     1     2     3     4     5

C{2} =
     6     7     8     9    10
    11    12    13    14    15
    16    17    18    19    20
```

See Also

cell2mat | num2cell

mat2str

Purpose

Convert matrix to string

Syntax

```
str = mat2str(A)
str = mat2str(A,n)
str = mat2str(A, 'class')
str = mat2str(A, n, 'class')
```

Description

`str = mat2str(A)` converts matrix `A` into a string. This string is suitable for input to the `eval` function such that `eval(str)` produces the original matrix to within 15 digits of precision.

`str = mat2str(A,n)` converts matrix `A` using `n` digits of precision.

`str = mat2str(A, 'class')` creates a string with the name of the class of `A` included. This option ensures that the result of evaluating `str` will also contain the class information.

`str = mat2str(A, n, 'class')` uses `n` digits of precision and includes the class information.

Limitations

The `mat2str` function is intended to operate on scalar, vector, or rectangular array inputs only. An error will result if `A` is a multidimensional array.

Examples

Example 1

Consider the matrix

```
x = [3.85 2.91; 7.74 8.99]
x =
    3.8500    2.9100
    7.7400    8.9900
```

The statement

```
A = mat2str(x)
```

produces

```
A =
   [3.85 2.91;7.74 8.99]
```

where `A` is a string of 21 characters, including the square brackets, spaces, and a semicolon.

`eval(mat2str(x))` reproduces `x`.

Example 2

Create a 1-by-6 matrix of signed 16-bit integers, and then use `mat2str` to convert the matrix to a 1-by-33 character array, `A`. Note that output string `A` includes the class name, `int16`:

```
x1 = int16([-300 407 213 418 32 -125]);

A = mat2str(x1, 'class')
A =
   int16([-300 407 213 418 32 -125])

class(A)
ans =
   char
```

Evaluating the string `A` gives you an output `x2` that is the same as the original `int16` matrix:

```
x2 = eval(A);

if isnumeric(x2) && isa(x2, 'int16') && all(x2 == x1)
   disp 'Conversion back to int16 worked'
end

Conversion back to int16 worked
```

See Also

`num2str` | `int2str` | `str2num` | `sprintf` | `fprintf`

material

Purpose

Control reflectance properties of surfaces and patches

Syntax

```
material shiny
material dull
material metal
material([ka kd ks])
material([ka kd ks n])
material([ka kd ks n sc])
material default
```

Description

`material` sets the lighting characteristics of `surface` and `patch` objects.

`material shiny` sets the reflectance properties so that the object has a high specular reflectance relative to the diffuse and ambient light, and the color of the specular light depends only on the color of the light source.

`material dull` sets the reflectance properties so that the object reflects more diffuse light and has no specular highlights, but the color of the reflected light depends only on the light source.

`material metal` sets the reflectance properties so that the object has a very high specular reflectance, very low ambient and diffuse reflectance, and the color of the reflected light depends on both the color of the light source and the color of the object.

`material([ka kd ks])` sets the ambient/diffuse/specular strength of the objects.

`material([ka kd ks n])` sets the ambient/diffuse/specular strength and specular exponent of the objects.

`material([ka kd ks n sc])` sets the ambient/diffuse/specular strength, specular exponent, and specular color reflectance of the objects.

`material default` sets the ambient/diffuse/specular strength, specular exponent, and specular color reflectance of the objects to their defaults.

Tips

The material command sets the AmbientStrength, DiffuseStrength, SpecularStrength, SpecularExponent, and SpecularColorReflectance properties of all surface and patch objects in the axes. There must be visible light objects in the axes for lighting to be enabled. Look at the materal.m file to see the actual values set (enter the command type material).

See Also

light | lighting | patch | surface

How To

- "Lighting Overview"

matfile

Purpose

Load and save parts of variables in MAT-files

Syntax

```
matObj = matfile(filename)
matObj = matfile(filename,'Writable',isWritable)
```

Description

`matObj = matfile(filename)` constructs a `matlab.io.MatFile` object that can load or save parts of variables in MAT-file `filename`. MATLAB does not load any data from the file into memory when creating the object.

`matObj = matfile(filename,'Writable',isWritable)` enables or disables write access to the file for object `matObj`. Possible values for `isWritable` are logical `true` (1) or `false` (0).

Input Arguments

filename

String enclosed in single quotation marks that specifies the name of a MAT-file.

`filename` can include a full or partial path, otherwise `matfile` searches for the file along the MATLAB search path. If `filename` does not include an extension, `matfile` appends `.mat`.

If the file does not exist, `matfile` creates a Version 7.3 MAT-file on the first assignment to a variable.

`matfile` only supports partial loading and saving for MAT-files in Version 7.3 format (described in MAT-File Versions). If you index into a variable in a Version 7 (the current default) or earlier MAT-file, MATLAB warns and temporarily loads the entire contents of the variable.

'Writable'

Parameter to use with the `isWritable` argument.

isWritable

Logical value that specifies whether to allow saving to the file. Possible values:

`true (1)`	Enable saving. If the file is read only, change the system permissions with `fileattrib`.
`false (0)`	Disable saving with `matfile`. MATLAB does not change the system permissions.

Default: `true` for new files, `false` for existing files

Output Arguments

matObj

Object of class `matlab.io.MatFile` that corresponds to a MAT-file.

Access variables in the MAT-file as properties of `matObj`, with dot notation similar to accessing fields of structs. The syntax for loading and saving part of variable `varName` in the MAT-file corresponding to `matObj` is:

```
loadedData = matObj.varName(indices);    % Load into workspace
matObj.varName(indices) = dataToSave;    % Save to file
```

When indexing, specify indices for all dimensions. Indices can be a single value, an equally spaced range of increasing values, or a colon (`:`), such as

```
matObj.varName(100:500, 200:600)
matObj.varName(:, 501:1000)
matObj.varName(1:2:1000, 80)
```

Limitations

- Using the `end` keyword as part of an index causes MATLAB to load the entire variable into memory. For very large variables, this load operation results in `Out of Memory` errors. Rather than using `end`, determine the extent of a variable with the `size` method, such as:

  ```
  sizeMyVar = size(matObj,'myVar')
  ```

- `matfile` does not support linear indexing. You must specify indices for all dimensions.

- `matfile` does not support indexing into:
  - Cells of cell arrays
  - Fields of structs
  - User-defined classes
  - Sparse arrays
- You cannot assign complex values to an indexed portion of a real array.
- You cannot evaluate function handles using a `MatFile` object. For example, if your MAT-file contains function handle `myfunc`, the syntax `matObj.myfunc()` attempts to index into the function handle, and does not invoke the function.

Examples

Create `myFile.mat` in a temporary folder and save data to part of variable `savedVar`:

```
filename = fullfile(tempdir,'myFile.mat');
matObj = matfile(filename);
matObj.savedVar(81:100,81:100) = magic(20);
```

Load part of the data into variable `loadVar`:

```
loadVar = matObj.savedVar(85:94,85:94);
```

Load or save an entire variable by omitting the indices. For example, load variable `topo` from `topography.mat`:

```
filename = 'topography.mat';
matObj = matfile(filename);
topo = matObj.topo;
```

Determine the dimensions of a variable, and process one part of the variable at a time. In this case, calculate and store the average of each column of variable `stocks` in the example file `stocks.mat`:

```
filename = 'stocks.mat';
matObj = matfile(filename);
[nrows, ncols] = size(matObj,'stocks');

avgs = zeros(1,ncols);
for idx = 1:ncols
    avgs(idx) = mean(matObj.stocks(:,idx));
end
```

By default, `matfile` only supports loading data from existing files. To enable saving, set `Writable` to `true` either during construction of the object,

```
filename = 'myFile.mat';
matObj = matfile(filename,'Writable',true);
```

or in a separate step, by setting `Properties.Writable`:

```
filename = 'myFile.mat';
matObj = matfile(filename);
matObj.Properties.Writable = true;
```

See Also

`load` | `save` | `size` | `whos`

matlab.io.MatFile

Purpose

Load and save parts of variables in MAT-files

Description

The `matfile` function constructs a `matlab.io.MatFile` object that corresponds to a MAT-File, such as

```
matObj = matfile('myFile.mat')
```

Access variables in the MAT-file as properties of `matObj`, with dot notation similar to accessing fields of structs. The syntax for loading and saving part of variable `varName` in the MAT-file corresponding to `matObj` is:

```
loadedData = matObj.varName(indices);    % Load into workspace
matObj.varName(indices) = dataToSave;    % Save to file
```

When indexing, specify indices for all dimensions. Indices can be a single value, an equally spaced range of increasing values, or a colon (`:`), such as

```
matObj.varName(100:500, 200:600)
matObj.varName(:, 501:1000)
matObj.varName(1:2:1000, 80)
```

Limitations

- Using the `end` keyword as part of an index causes MATLAB to load the entire variable into memory. For very large variables, this load operation results in `Out of Memory` errors. Rather than using `end`, determine the extent of a variable with the `size` method, such as:

  ```
  sizeMyVar = size(matObj,'myVar')
  ```

- `matfile` does not support linear indexing. You must specify indices for all dimensions.
- `matfile` does not support indexing into:
  - Cells of cell arrays
  - Fields of structs

  - User-defined classes
  - Sparse arrays
- You cannot assign complex values to an indexed portion of a real array.
- You cannot evaluate function handles using a `MatFile` object. For example, if your MAT-file contains function handle `myfunc`, the syntax `matObj.myfunc()` attempts to index into the function handle, and does not invoke the function.

Construction

`matObj = matfile(filename)` constructs a `matlab.io.MatFile` object that can load or save parts of variables in MAT-file `filename`. MATLAB does not load any data from the file into memory when creating the object.

`matObj = matfile(filename,'Writable',isWritable)` enables or disables write access to the file for object `matObj`. Possible values for `isWritable` are logical `true` (`1`) or `false` (`0`).

Input Arguments

filename

String enclosed in single quotation marks that specifies the name of a MAT-file.

`filename` can include a full or partial path, otherwise `matfile` searches for the file along the MATLAB search path. If `filename` does not include an extension, `matfile` appends `.mat`.

If the file does not exist, `matfile` creates a Version 7.3 MAT-file on the first assignment to a variable.

`matfile` only supports partial loading and saving for MAT-files in Version 7.3 format (described in MAT-File Versions). If you index into a variable in a Version 7 (the current default) or earlier MAT-file, MATLAB warns and temporarily loads the entire contents of the variable.

'Writable'

Parameter to use with the `isWritable` argument.

isWritable

Logical value that specifies whether to allow saving to the file. Possible values:

`true` (1)	Enable saving. If the file is read only, change the system permissions with `fileattrib`.
`false` (0)	Disable saving with `matfile`. MATLAB does not change the system permissions.

Default: `true` for new files, `false` for existing files

Properties

Properties.Source

String that contains the fully qualified path to the file. Read only.

Properties.Writable

Logical value that specifies whether to allow saving to the file. Possible values:

`true` (1)	Enable saving. If the file is read only, change the system permissions with `fileattrib`.
`false` (0)	Disable saving with `matfile`. MATLAB does not change the system permissions.

Default: `true` for new files, `false` for existing files

Methods

size	Array dimensions
who	Names of variables in MAT-file
whos	Names, sizes, and types of variables in MAT-file

You cannot access help for these methods using the `help` command. Find help on the methods from the command line using the `doc` command, such as `doc matlab.io.MatFile/size`.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create `myFile.mat` in a temporary folder and save data to part of variable `savedVar`:

```
filename = fullfile(tempdir,'myFile.mat');
matObj = matfile(filename);
matObj.savedVar(81:100,81:100) = magic(20);
```

Load part of the data into variable `loadVar`:

```
loadVar = matObj.savedVar(85:94,85:94);
```

Load or save an entire variable by omitting the indices. For example, load variable `topo` from `topography.mat`:

```
filename = 'topography.mat';
matObj = matfile(filename);
topo = matObj.topo;
```

Determine the dimensions of a variable, and process one part of the variable at a time. In this case, calculate and store the average of each column of variable `stocks` in the example file `stocks.mat`:

```
filename = 'stocks.mat';
matObj = matfile(filename);
[nrows, ncols] = size(matObj,'stocks');

avgs = zeros(1,ncols);
for idx = 1:ncols
```

```
    avgs(idx) = mean(matObj.stocks(:,idx));
end
```

By default, `matfile` only supports loading data from existing files. To enable saving, set `Writable` to `true` either during construction of the object,

```
filename = 'myFile.mat';
matObj = matfile(filename,'Writable',true);
```

or in a separate step, by setting `Properties.Writable`:

```
filename = 'myFile.mat';
matObj = matfile(filename);
matObj.Properties.Writable = true;
```

See Also

`load` | `save` | `size` | `whos`

Purpose

Startup file for MATLAB program

Description

At startup time, MATLAB automatically executes the file `matlabrc.m`. This function establishes the MATLAB path, sets the default figure size, and sets some uicontrol defaults.

On multiuser or networked systems, system administrators can put any messages, definitions, or other code that applies to all users in their `matlabrc.m` file.

The file `matlabrc.m` invokes the `startup.m` file, if it exists on the search path MATLAB uses.

Individual users should use the `startup.m` file to customize MATLAB startup. The `matlabrc.m` file is reserved for system administrators.

Algorithms

MATLAB invokes `matlabrc` at startup. `matlabrc.m` contains the statements

```
if exist('startup') == 2
    startup
end
```

that invokes `startup.m`, if it exists. Extend this process to create additional startup files, if required.

Tips

You can also start MATLAB using options you define at the Command Window prompt or in your Microsoft Windows shortcut for MATLAB.

Examples

Turning Off the Figure Window Toolbar

If you do not want the toolbar to appear in the figure window, remove the comment marks from the following line in the `matlabrc.m` file, or create a similar line in your own `startup.m` file.

```
%   set(0,'defaultfiguretoolbar','none')
```

See Also

`matlabroot` | `quit` | `restoredefaultpath` | `startup`

How To

- Startup Options

Purpose

Root folder

Syntax

```
matlabroot
mr = matlabroot
```

Description

`matlabroot` returns the name of the folder where the MATLAB software is installed. Use `matlabroot` to create a path to MATLAB and toolbox folders that does not depend on a specific platform, MATLAB version, or installation location.

`mr = matlabroot` returns the name of the folder in which the MATLAB software is installed and assigns it to `mr`.

Tips

matlabroot as Folder Name

The term *`matlabroot`* also refers to the folder where MATLAB files are installed. For example, "save to *`matlabroot`*`/toolbox/local`" means save to the `toolbox/local` folder in the MATLAB root folder.

Using $matlabroot as a Literal

In some files, `$matlabroot` is literal. In those files, MATLAB interprets `$matlabroot` as the full path to the MATLAB root folder. For example, including the line:

```
$matlabroot/toolbox/local/myfile.jar
```

in `javaclasspath.txt`, adds `myfile.jar`, which is located in the `toolbox/local` folder, to the static Java class path.

Sometimes, particularly in older code examples, the term `$matlabroot` or `$MATLABROOT` is not meant to be interpreted literally but is used to represent the value returned by the `matlabroot` function.

matlabroot on Macintosh Platforms

In R2008b (V7,7) and more recent versions, running `matlabroot` on Apple Macintosh platforms returns

```
/Applications/MATLAB_R2008b.app
```

In versions prior to R2008b (V7.7), such as R2008a (V7.6), running `matlabroot` on Macintosh platforms returns, for example

```
/Applications/MATLAB_R2008a
```

When you use GUIs on Macintosh platforms, you cannot directly view the contents of the MATLAB root folder. For more information, see "Navigating Within the MATLAB Root Folder on Macintosh Platforms".

Examples

Get the location where MATLAB is installed:

```
matlabroot
```

MATLAB returns:

```
C:\Program Files\MATLAB\R2009a
```

Produce a full path to the `toolbox/matlab/general` folder that is correct for the platform on which it is executed:

```
fullfile(matlabroot,'toolbox','matlab','general')
```

Change the current folder to the MATLAB root folder:

```
cd(matlabroot)
```

To add the folder `myfiles` to the MATLAB search path, run

```
addpath([matlabroot '/toolbox/local/myfiles'])
```

See Also

`fullfile` | `path` | `toolboxdir`

How To

- "Understanding File Locations in MATLAB"

Purpose

Start MATLAB program (UNIX platforms)

Syntax

```
matlab helpOption
matlab envDispOption
matlab archOption
matlab dispOption
matlab modeOption
matlab -c licensefile
matlab -debug
matlab -Ddebugger options
matlab -jdb portnumber
matlab -logfile filename
matlab -mwvisual visualid
matlab -noFigureWindows
matlab -nosplash
matlab -nouserjavapath
matlab -r "command"
matlab -singleCompThread
```

Note You can enter more than one of these options in the same `matlab` command. If you use **`-D`**`debugger` to start MATLAB in debug mode, the first option in the command must be **`-D`**`debugger`.

Description

`matlab` is a Bourne shell script that starts the MATLAB executable on UNIX platforms. (In this document, `matlab` refers to this script; MATLAB refers to the application program). Before actually initiating the execution of MATLAB, this script configures the run-time environment by:

- Determining the MATLAB root folder
- Determining the host machine architecture
- Processing any command line options
- Reading the MATLAB startup file, `.matlab7rc.sh`

- Setting MATLAB environment variables

There are two ways in which you can control the way the `matlab` script works:

- By specifying command line options
- By assigning values in the MATLAB startup file, `.matlab7rc.sh`

Specifying Options at the Command Line

Note On Apple Macintosh platforms, the `matlab` script is located inside the MATLAB application package:

```
/Applications/MATLAB_R2012b.app/bin/matlab
```

To run the `matlab` script from Terminal, specify the full path as shown above, or `cd` to the `bin` directory within the application package and preface the command with `./` characters. For example:

```
./matlab -nojvm -nodisplay -nosplash -r "myfun(10,30)"
```

Options that you can enter at the command line are as follows:

`matlab helpOption` displays information that matches the specified `helpOption` argument without starting MATLAB. `helpOption` can be any one of the keywords shown in the table below. Enter only one `helpOption` keyword in a `matlab` command.

Values for helpOption

Option	Description
`-help`	Display `matlab` command usage.
`-h`	The same as **`-help`**.

`matlab envDispOption` displays the values of environment variables passed to MATLAB or their values just prior to exiting MATLAB. `envDispOption` can be either one of the options shown in the table below.

Values for envDispOption

Option	Description
-n	Display all the final values of the environment variables and arguments passed to the MATLAB executable as well as other diagnostic information. Does not start MATLAB.
-e	Display *all* environment variables and their values just prior to exiting. This argument must have been parsed before exiting for anything to be displayed. The last possible exiting point is just before the MATLAB image would have been executed and a status of `0` is returned. If the exit status is not `0` on return, then the variables and values may not be correct. Does not start MATLAB.

`matlab archOption` starts MATLAB and assumes that you are running on the system architecture specified by `arch`, or using the MATLAB version specified by `variant`, or both. The values for the `archOption` argument are shown in the table below. Enter only one of these options in a `matlab` command.

Values for archOption

Option	Description
`-arch`	Run MATLAB assuming this architecture rather than the actual architecture of the machine you are using. Replace the term `arch` with a string representing a recognized system architecture. For example: `matlab -glnx86`
`v=variant`	Execute the version of MATLAB found in the folder `bin/$ARCH/variant` instead of `bin/$ARCH`. Replace the term `variant` with a string representing a MATLAB version.
`v=arch/variant`	Execute the version of MATLAB found in the folder `bin/arch/variant` instead of `bin/$ARCH`. Replace the terms `arch` and `variant` with strings representing a specific architecture and MATLAB version.

`matlab dispOption` starts MATLAB using one of the display options shown in the table below. Enter only one of these options in a `matlab` command.

Values for dispOption

Option	Description
`-display` `xDisp`	Send X commands to X Window Server display `xDisp`. This supersedes the value of the `DISPLAY` environment variable. On Macintosh platforms, this option is ignored.
`-nodisplay`	Start the Oracle JVM software, but do not start the MATLAB desktop. Do not display

(Continued)

Option	Description
	any X commands, and ignore the `DISPLAY` environment variable. On Macintosh platforms, start the JVM in headless mode, and prevent all windows from being displayed.

`matlab modeOption` starts MATLAB without its usual desktop component. Enter only one of the options shown below.

Values for modeOption

Option	Description
`-desktop`	Allow the MATLAB desktop to be started by a process without a controlling terminal. This is usually a required command line argument when attempting to start MATLAB from a window manager menu or desktop icon.
`-nodesktop`	Start MATLAB without bringing up the MATLAB desktop. The JVM software *is* started. Use the current window in the operating system to enter commands. Use this option to run in batch processing mode. Note that if you pipe to MATLAB using the `>` constructor, the `nodesktop` option is used automatically. With `nodesktop`, MATLAB does not save statements to the Command History. With `nodesktop`, you can still use most development environment tools by starting them via a function. For example, use `preferences` to open the Preferences dialog box and `doc` to open the Help browser. Do not use `nodesktop` to provide a Command

(Continued)

Option	Description
	Window-only interface; instead, select **Desktop > Desktop Layout > Command Window Only**.
`-nojvm`	Start MATLAB without the JVM software. Use the current window to enter commands. The MATLAB desktop does not open. Any tools that require Java software, such as the desktop tools, cannot be used. Handle Graphics and related functionality are not supported; MATLAB produces a warning when you use them.

`matlab -c licensefile` starts MATLAB using the specified license file. The `licensefile` argument can have the form `port@host` or it can be a colon-separated list of license filenames. This option causes the `LM_LICENSE_FILE` and `MLM_LICENSE_FILE` environment variables to be ignored.

`matlab -debug` starts MATLAB and displays debugging information that can be useful, especially for X based problems. This option should be used only when working with a Technical Support Representative from The MathWorks, Inc.

`matlab -Ddebugger options` starts MATLAB in debug mode, using the named `debugger` (for example, `dbx`, `gdb`, `xdb`, `cvd`, `lldb`). A full path can be specified for debugger.

Notes for -Ddebugger Argument

- The `options` argument can include *only* those options that follow the debugger name in the syntax of the actual debug command. For most debuggers, there is a very limited number of such options. Options that would normally be passed to the MATLAB executable should be

used as parameters of a command inside the debugger (like `run`). They should not be used when running the `matlab` script.

- If any other `matlab` command options are placed before the **-D**`debugger` argument, they will be handled as if they were part of the options after the **-D**`debugger` argument and will be treated as illegal options by most debuggers. The `MATLAB_DEBUG` environment variable is set to the filename part of the debugger argument.
- To customize your debugging session, use a startup file. See your debugger documentation for details.
- For certain debuggers like `gdb`, the `SHELL` environment variable is *always* set to `/bin/sh`.

`matlab` **`-jdb`** `portnumber` starts MATLAB, enabling use of the Java debugger. If you specify a port number, the Java debugger uses that port to communicate with MATLAB. You can specify any port number in the range 0-65535 that is not reserved or currently in use by another application on your system. By default, MATLAB uses port 4444. If you are running multiple MATLAB sessions and want to use the Java debugger, be sure to specify a port number.

`matlab` **`-logfile`** `filename` starts MATLAB and makes a copy of any output to the command window in file log. This includes all crash reports.

`matlab` **`-mwvisual`** `visualid` starts MATLAB and uses `visualid` as the default X visual for figure windows. `visualid` is a hexadecimal number that can be found using `xdpyinfo`.

`matlab` **`-noFigureWindows`** starts MATLAB, but disables the display of any figure windows in MATLAB.

`matlab` **`-nosplash`** starts MATLAB but does not display the splash screen during startup.

`matlab` **`-nouserjavapath`** disables `javaclasspath.txt` and `javalibrarypath.txt` files.

`matlab` **`-r`** `"command"` starts MATLAB and executes the specified MATLAB `command`. Include the command in double quotation marks

("*command*"). If command is the name of a MATLAB function or script, do not specify the file extension. Use semicolons or commas to separate multiple statements.

matlab **-singleCompThread** limits MATLAB to a single computational thread. By default, MATLAB makes use of the multithreading capabilities of the computer on which it is running.

Specifying Options in the MATLAB Startup File

The .matlab7rc.sh shell script contains definitions for a number of variables that the matlab script uses. These variables are defined within the matlab script, but can be redefined in .matlab7rc.sh. When invoked, matlab looks for the first occurrence of .matlab7rc.sh in the current folder, in the home folder ($HOME), and in the *matlabroot*/bin folder, where the template version of .matlab7rc.sh is located.

You can edit the template file to redefine information used by the matlab script. If you do not want your changes applied systemwide, copy the edited version of the script to your current or home folder. Ensure that you edit the section that applies to your machine architecture.

The following table lists the variables defined in the.matlab7rc.sh file. See the comments in the .matlab7rc.sh file for more information about these variables.

Variable Options for Startup File

Variable	Definition and Standard Assignment Behavior
ARCH	The machine architecture. The value ARCH passed with the -arch or -arch/ext argument to the script is tried first, then the value of the environment variable MATLAB_ARCH is tried next, and finally it is

Variable	Definition and Standard Assignment Behavior
	computed. The first one that gives a valid architecture is used.
`DISPLAY`	Linux platforms only. The hostname of the X Window display MATLAB uses for output. The value of `Xdisplay` passed with the `-display` argument to the script is used; otherwise, the value in the environment is used. `DISPLAY` is ignored by MATLAB if the `-nodisplay` argument is passed.
`LD_LIBRARY_PATH` on Linux platforms. `DYLD_LIBRARY_PATH` on Macintosh platforms.	Final Load library path. The name `LD_LIBRARY_PATH` is platform dependent. The final value is normally a colon-separated list of four sublists, each of which could be empty. The first sublist is defined in `.matlab7rc.sh` as `LDPATH_PREFIX`. The second sublist is computed in the script and includes folders inside the MATLAB root folder and relevant Java folders. The third sublist contains any nonempty value of `LD_LIBRARY_PATH` from the environment possibly augmented in `.matlab7rc.sh`. The final sublist is defined in `.matlab7rc.sh` as `LDPATH_SUFFIX`.

Variable	Definition and Standard Assignment Behavior
`LM_LICENSE_FILE`	The FLEX lm license variable. The license file value passed with the `-c` argument to the script is used; otherwise it is the value set in `.matlab7rc.sh`. In general, the final value is a colon-separated list of license files and/or `port@host` entries. The shipping `.matlab7rc.sh` file starts out the value by prepending `LM_LICENSE_FILE` in the environment to a default `license.file`. Later in the `matlab` script, if the `-c` option is not used, the *matlabroot*`/etc` folder is searched for the files that start with `license.dat.DEMO`. These files are assumed to contain demo licenses and are added automatically to the end of the current list.
`MATLAB`	The MATLAB root folder. The default computed by the script is used unless `MATLABdefault` is reset in `.matlab7rc.sh`. Currently `MATLABdefault` is not reset in the shipping `.matlab7rc.sh`.
`MATLAB_DEBUG`	Normally set to the name of the debugger. The `-Ddebugger` argument passed to the script sets this variable. Otherwise, a nonempty value in the environment is used.

Variable	Definition and Standard Assignment Behavior
MATLAB_JAVA	The path to the root of the Java Runtime Environment. The default set in the script is used unless MATLAB_JAVA is already set. Any nonempty value from .matlab7rc.sh is used first, then any nonempty value from the environment. Currently there is no value set in the shipping .matlab67rc.sh, so that environment alone is used.
MATLABPATH	The MATLAB search path. The final value is a colon-separated list with the MATLABPATH from the environment prepended to a list of computed defaults. You can add subfolders of userpath to theMATLAB search path upon startup. See userpath for details.
SHELL	The shell to use when the "!" or unix command is issued in MATLAB. This is taken from the environment unless SHELL is reset in .matlab7rc.sh. Note that an additional environment variable called MATLAB_SHELL takes precedence over SHELL. MATLAB checks internally for MATLAB_SHELL first and, if empty or not defined, then checks SHELL. If SHELL is also empty or not defined, MATLAB uses /bin/sh. The value of MATLAB_SHELL should be an absolute path, i.e. /bin/sh, not simply sh. Currently, the shipping .matlab7rc.sh file does not reset SHELL and also does not reference or set MATLAB_SHELL.

Variable	Definition and Standard Assignment Behavior
TOOLBOX	Path of the toolbox folder. A nonempty value in the environment is used first. Otherwise, *matlabroot*/toolbox, computed by the script, is used unless TOOLBOX is reset in .matlab7rc.sh. Currently TOOLBOX is not reset in the shipping .matlab7rc.sh.
XAPPLRESDIR	The X application resource folder (Linux systems only). A nonempty value in the environment is used first unless XAPPLRESDIR is reset in .matlab7rc.sh.
XKEYSYMDB	The X keysym database file (Linux systems only). A nonempty value in the environment is used first unless XKEYSYMDB is reset in .matlab7rc.sh.

The matlab script determines the path of the MATLAB root folder by looking up the folder tree from the *matlabroot*/bin folder (where the matlab script is located). The MATLAB variable is then used to locate all files within the MATLAB folder tree.

You can change the definition of MATLAB if, for example, you want to run a different version of MATLAB or if, for some reason, the path determined by the matlab script is not correct. (This can happen when certain types of automounting schemes are used by your system.)

See Also

matlab (Windows) | mex

How To

- "Start MATLAB on Linux Platforms"
- "Start MATLAB on Macintosh Platforms"

- “Startup Options”

matlab (Windows)

Purpose

Start MATLAB program (Windows platforms)

Syntax

```
matlab helpOption
matlab -automation
matlab -c licensefile
matlab -jdb portnumber
matlab -logfile filename
matlab -noFigureWindows
matlab -nosplash
matlab -nouserjavapath
matlab -r "statement"
matlab -regserver
matlab -sd "startdir"
matlab shieldOption
matlab -singleCompThread
matlab -unregserver
matlab -wait
```

Note You can enter more than one of these options in the same `matlab` command.

Description

`matlab` is a script that runs the main MATLAB executable on Microsoft Windows platforms. (In this document, the term `matlab` refers to the script, and MATLAB refers to the main executable). Before actually initiating the execution of MATLAB, it configures the run-time environment by:

- Determining the MATLAB root folder.
- Determining the host machine architecture.
- Selectively processing command line options with the rest passed to MATLAB.
- Setting certain MATLAB environment variables.

There are two ways in which you can control the way `matlab` works:

- By specifying command line options.
- By setting environment variables before calling the program.

Specifying Options at the Command Line

Options that you can enter at the command line are as follows:

`matlab` *`helpOption`* displays information that matches the specified *`helpOption`* argument without starting MATLAB. *`helpOption`* can be any one of the keywords shown in the table below. Enter only one *`helpOption`* keyword in a `matlab` statement.

Values for helpOption

Option	Description
`-help`	Display `matlab` command usage.
`-h`	The same as **`-help`**.
`-?`	The same as **`-help`**.

`matlab` **`-automation`** starts MATLAB as an automation server. The server window is minimized, and the MATLAB splash screen does not display on startup.

`matlab` **`-c`** `licensefile` starts MATLAB using the specified license file. The `licensefile` argument can have the form `port@host`. This option causes MATLAB to ignore the `LM_LICENSE_FILE` and `MLM_LICENSE_FILE` environment variables.

`matlab` **`-jdb`** `portnumber` starts MATLAB, enabling use of the Java debugger. If you specify a port number, the Java debugger uses that port to communicate with MATLAB. You can specify any port number in the range 0-65535 that is not reserved or currently in use by another application on your system. By default, MATLAB uses port 4444. If you are running multiple MATLAB sessions and want to use the Java debugger, be sure to specify a port number.

`matlab` **`-logfile`** `filename` starts MATLAB and makes a copy of any output to the Command Window in `filename`. This includes all crash reports.

`matlab` **`-noFigureWindows`** starts MATLAB, but disables the display of any figure windows in MATLAB.

`matlab` **`-nosplash`** starts MATLAB, but does not display the splash screen during startup.

`matlab` **`-nouserjavapath`** disables `javaclasspath.txt` and `javalibrarypath.txt` files.

`matlab` **`-r`** `"statement"` starts MATLAB and executes the specified MATLAB `statement`. If `statement` is the name of a MATLAB function or script, do not specify the file extension. Any required file must be on the MATLAB search path or in the startup folder.

`matlab` **`-regserver`** registers MATLAB as a Component Object Model (COM) server.

`matlab` **`-sd`** `"`*`startdir`*`"` specifies the startup folder for MATLAB (the current folder in MATLAB after startup). The **`-sd`** option has been deprecated. For information about alternatives, see "Startup Folder on Windows Platforms".

`matlab` *`shieldOption`* provides the specified level of protection of the address space used by MATLAB during startup on Windows 32–bit platforms. It attempts to help ensure the largest contiguous block of memory available after startup, which is useful for processing large data sets. The *`shieldOption`* does this by ensuring resources such as DLLs, are loaded into locations that will not fragment the address space. With *`shieldOption`* set to a value other than `none`, address space is protected up to or after the processing of `matlabrc`. Use higher levels of protection to secure larger initial blocks of contiguous memory, however a higher level might not always provide a larger size block and might cause startup problems. Therefore, start with a lower level of protection, and if successful, try the next higher level. You can use the `memory` function after startup to see the size of the largest contiguous block of memory; this helps you determine the actual effect of the *`shieldOption`* setting

you used. If your `matlabrc` (or `startup.m`) requires significant memory, a higher level of protection for *`shieldOption`* might cause startup to fail; in that event try a lower level. Values for *`shieldOption`* can be any one of the keywords shown in the table below.

Option	Description
`-shield minimum`	This is the default setting. It protects the range `0x50000000` to `0x70000000` during MATLAB startup until just before startup processes `matlabrc`. It ensures there is at least approximately 500 MB of contiguous memory up to this point. Start with this, the default value. To use the default, do not specify a shield option upon startup. If MATLAB fails to start successfully using the default option, **`-shield minimum`**, instead use **`-shield none`**. If MATLAB starts successfully with the default value for *`shieldOption`* and you want to try to ensure an even larger contiguous block after startup, try using the **`-shield medium`** option.
`-shield medium`	This protects the same range as for minimum, `0x50000000` to `0x70000000`, but protects the range until just after startup processes `matlabrc`. It ensures there is at least approximately 500 MB of contiguous memory up to this point. If MATLAB fails to start successfully with the **`-shield medium`** option, instead use the default option (**`-shield minimum`**).

Option	Description
	If MATLAB starts successfully with the **-shield medium** option and you want to try to ensure an even larger contiguous block after startup, try using the **-shield maximum** option.
-shield maximum	This protects the maximum possible range, which can be up to approximately 1.5 GB, until just after startup processes `matlabrc`. If MATLAB fails to start successfully with the **-shield maximum** option, instead use the **-shield medium** option.
-shield none	This completely disables address shielding. Use this if MATLAB fails to start successfully with the default (**-shield minimum**) option.

`matlab` **`-singleCompThread`** limits MATLAB to a single computational thread. By default, MATLAB makes use of the multithreading capabilities of the computer on which it is running.

`matlab` **`-unregserver`** removes all MATLAB COM server entries from the registry.

`matlab` **`-wait`** MATLAB is started by a separate starter program which normally launches MATLAB and then immediately quits. Using this option tells the starter program not to quit until MATLAB has terminated. This option is useful when you need to process the results from MATLAB in a script. Calling MATLAB with this option blocks the script from continuing until the results are generated.

Setting Environment Variables

You can set the following environment variables before starting MATLAB.

Variable Name	Description
`LM_LICENSE_FILE`	This variable specifies the License File to use. If you use the **`-c`** argument to specify the License File it overrides this variable. The value of this variable can be a list of License Files, separated by semi-colons, or `port@host` entries.

See Also

`matlab (UNIX)` | `mex` | `userpath`

How To

- "Start MATLAB on Windows Platforms"
- "Startup Options"

Purpose

Largest elements in array

Syntax

```
C = max(A)
C = max(A,B)
C = max(A,[],dim)
[C,I] = max(...)
```

Description

`C = max(A)` returns the largest elements along different dimensions of an array.

If `A` is a vector, `max(A)` returns the largest element in `A`.

If `A` is a matrix, `max(A)` treats the columns of `A` as vectors, returning a row vector containing the maximum element from each column.

If `A` is a multidimensional array, `max(A)` treats the values along the first non-singleton dimension as vectors, returning the maximum value of each vector.

`C = max(A,B)` returns an array the same size as `A` and `B` with the largest elements taken from `A` or `B`. The dimensions of `A` and `B` must match, or they may be scalar.

`C = max(A,[],dim)` returns the largest elements along the dimension of `A` specified by scalar `dim`. For example, `max(A,[],1)` produces the maximum values along the first dimension of `A`.

`[C,I] = max(...)` finds the indices of the maximum values of `A`, and returns them in output vector `I`. If there are several identical maximum values, the index of the first one found is returned.

Examples

Return the maximum of a 2-by-3 matrix from each column:

```
X = [2 8 4; 7 3 9];
max(X,[],1)
ans =

     7     8     9
```

Return the maximum from each row:

```
max(X,[],2)
ans =

     8
     9
```

Compare each element of X to a scalar:

```
max(X,5)
ans =

     5     8     5
     7     5     9
```

Tips

For complex input A, max returns the complex number with the largest complex modulus (magnitude), computed with max(abs(A)). Then computes the largest phase angle with max(angle(x)), if necessary.

The max function ignores NaNs.

See Also

isnan | mean | median | min | sort

MaximizeCommandWindow

Purpose

Open Automation server window

Syntax

MATLAB Client

```
h.MaximizeCommandWindow
MaximizeCommandWindow(h)
```

IDL Method Signature

```
HRESULT MaximizeCommandWindow(void)
```

Microsoft Visual Basic Client

```
MaximizeCommandWindow
```

Description

`h.MaximizeCommandWindow` displays the window for the server attached to handle `h`, and makes it the currently active window on the desktop.

`MaximizeCommandWindow(h)` is an alternate syntax.

`MaximizeCommandWindow` restores the window to the size it had at the time it was minimized, not to the maximum size on the desktop. If the server window was not previously in a minimized state, `MaximizeCommandWindow` does nothing.

Examples

From a MATLAB client, modify the size of the command window in a MATLAB Automation server:

```
h = actxserver('matlab.application');
h.MinimizeCommandWindow;
% Now return the server window to its former state on
% the desktop and make it the currently active window.
h.MaximizeCommandWindow;
```

From a Visual Basic .NET client, modify the size of the command window in a MATLAB Automation server:

```
Dim Matlab As Object
```

```
Matlab = CreateObject("matlab.application")
Matlab.MinimizeCommandWindow

'Now return the server window to its former state on
'the desktop and make it the currently active window.

Matlab.MaximizeCommandWindow
```

See Also

`MinimizeCommandWindow`

How To

- "MATLAB COM Automation Server Interface"
- "Controlling the Server Window"

maxNumCompThreads

Purpose

Control maximum number of computational threads

Note `maxNumCompThreads` will be removed in a future version. You can set the `-singleCompThread` option when starting MATLAB to limit MATLAB to a single computational thread. By default, MATLAB makes use of the multithreading capabilities of the computer on which it is running.

Syntax

```
N = maxNumCompThreads
LASTN = maxNumCompThreads(N)
LASTN = maxNumCompThreads('automatic')
```

Description

`N = maxNumCompThreads` returns the current maximum number of computational threads `N`.

`LASTN = maxNumCompThreads(N)` sets the maximum number of computational threads to `N`, and returns the previous maximum number of computational threads, `LASTN`.

`LASTN = maxNumCompThreads('automatic')` sets the maximum number of computational threads using what the MATLAB software determines to be the most desirable. It additionally returns the previous maximum number of computational threads, `LASTN`.

Currently, the maximum number of computational threads is equal to the number of computational cores on your machine.

Note Setting the maximum number of computational threads using `maxNumCompThreads` does not propagate to your next MATLAB session.

Purpose

Average or mean value of array

Syntax

```
M = mean(A)
M = mean(A,dim)
```

Description

`M = mean(A)` returns the mean values of the elements along different dimensions of an array.

If `A` is a vector, `mean(A)` returns the mean value of `A`.

If `A` is a matrix, `mean(A)` treats the columns of `A` as vectors, returning a row vector of mean values.

If `A` is a multidimensional array, `mean(A)` treats the values along the first non-singleton dimension as vectors, returning an array of mean values.

`M = mean(A,dim)` returns the mean values for elements along the dimension of `A` specified by scalar `dim`. For matrices, `mean(A,2)` is a column vector containing the mean value of each row.

Examples

```
A = [1 2 3; 3 3 6; 4 6 8; 4 7 7];
mean(A)
ans =
    3.0000    4.5000    6.0000

mean(A,2)
ans =
    2.0000
    4.0000
    6.0000
    6.0000
```

See Also

`corrcoef` | `cov` | `max` | `median` | `min` | `mode` | `std` | `var`

median

Purpose

Median value of array

Syntax

```
M = median(A)
M = median(A,dim)
```

Description

`M = median(A)` returns the median value of `A`.

- If `A` is a vector, then `median(A)` returns the median value of `A`.
- If `A` is a nonempty matrix, then `median(A)` treats the columns of `A` as vectors and returns a row vector of median values.
- If `A` is an empty 0-by-0 matrix, `median(A)` returns `NaN`.
- If `A` is a multidimensional array, then `median(A)` acts along the first nonsingleton dimension and returns an array of median values. The size of this dimension reduces to `1` while the sizes of all other dimensions remain the same.

`median` computes natively in the numeric class of `A`, such that `class(M) = class(A)`.

`M = median(A,dim)` returns the median of elements along dimension `dim`. For example, if `A` is a matrix, then `median(A,2)` is a column vector containing the median value of each row.

Input Arguments

A - Input array

vector | matrix | multidimensional array

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

If `A` contains `NaN`, then `M` returns `NaN`.

Data Types

`double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

dim - Dimension to operate along

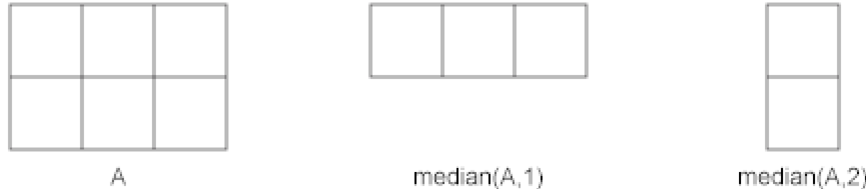
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, the default is the first nonsingleton dimension.

Dimension `dim` indicates the dimension whose length reduces to `1`. The `size(M,dim)` is `1`, while the sizes of all other dimensions remain the same.

Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `median(A,1)` returns a row vector containing the product of the elements in each column.
- If `dim = 2`, then `median(A,2)` returns a column vector containing the product of the elements in each row.



`median` returns `A` if `dim` is greater than `ndims(A)`.

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

Definitions

First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to `1`.

For example:

- If `X` is a 1-by-n row vector, then the second dimension is the first nonsingleton dimension of `X`.
- If `X` is a 1-by-0-by-n empty array, then the second dimension is the first nonsingleton dimension of `X`.

- If X is a 1-by-1-by-3 array, then the third dimension is the first nonsingleton dimension of X.

Examples

Median of Matrix Columns

Define a 4-by-3 matrix.

```
A = [0 1 1; 2 3 2; 1 3 2; 4 2 2]
```

```
A =

     0     1     1
     2     3     2
     1     3     2
     4     2     2
```

Find the median value of each column.

```
M = median(A)
```

```
M =

    1.5000    2.5000    2.0000
```

For each column, the median value is the mean of the middle two numbers in sorted order.

Median of Matrix Rows

Define a 2-by-3 matrix.

```
A = [0 1 1; 2 3 2]
```

```
A =

     0     1     1
     2     3     2
```

Find the median value of each row.

```
M = median(A,2)
```

```
M =

     1
     2
```

For each row, the median value is the middle number in sorted order.

Median of 3-D Array

Create a 1-by-3-by-4 array of integers between 1 and 10.

```
A = gallery('integerdata',10,[1,3,4],1)
```

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

    10     8    10


A(:,:,2) =

     6     9     5


A(:,:,3) =

     9     6     1


A(:,:,4) =

     4     9     5
```

Find the median values of this 3-D array along the second dimension.

```
M = median(A)
```

```
M(:,:,1) =
```

```
    10


M(:,:,2) =

     6


M(:,:,3) =

     6


M(:,:,4) =

     5
```

This operation produces a 1-by-1-by-4 array by computing the median of the three values along the second dimension. The size of the second dimension is reduced to `1`.

Compute the median along the first dimension of `A`.

```
M = median(A,1);
isequal(A,M)

ans =

     1
```

This returns the same array as `A` because the size of the first dimension is `1`.

Median of 8-bit Integer Array

Define a 1-by-4 vector of 8-bit integers.

```
A = int8(1:4)

A =

    1    2    3    4
```

Compute the median value.

```
M = median(A),
class(M)

M =

    3


ans =

int8
```

`M` is the mean of the middle two numbers in sorted order returned as an 8-bit integer.

See Also

`corrcoef` | `cov` | `max` | `mean` | `min` | `mode` | `std` | `var`

Purpose

Construct memmapfile object

Syntax

```
m = memmapfile(filename)
m = memmapfile(filename, prop1, value1, prop2, value2, ...)
```

Description

`m = memmapfile(filename)` constructs an object of the `memmapfile` class that maps file `filename` to memory using the default property values. The `filename` input is a quoted string that specifies the path and name of the file to be mapped into memory. `filename` must include a filename extension if the name of the file being mapped has an extension. The `filename` argument cannot include any wildcard characters (e.g., `*` or `?`), is case sensitive on The Open Group UNIX platforms, but is not case sensitive on Microsoft Windows platforms.

`m = memmapfile(filename, prop1, value1, prop2, value2, ...)` constructs an object of the `memmapfile` class that maps file `filename` into memory and sets the properties of that object that are named in the argument list (`prop1`, `prop2`, etc.) to the given values (`value1`, `value2`, etc.). All property name arguments must be quoted strings (e.g., `'Writable'`). Any properties that are not specified are given their default values.

Optional properties are shown in the table below and are described in the sections that follow.

Property	Description	Data Type	Default
`Format`	Format of the contents of the mapped region, including data type, array shape, and variable or field name by which	`char` array or N-by-3 `cell` array	`uint8`

Property	Description	Data Type	Default
	to access the data		
`Offset`	Number of bytes from the start of the file to the start of the mapped region. This number is zero-based. That is, offset 0 represents the start of the file.	`double`	`0`
`Repeat`	Number of times to apply the specified format to the mapped region of the file	`double`	`Inf`
`Writable`	Type of access allowed to the mapped region	`logical`	false

There are three different ways you can specify a value for the `Format` property. See the following sections in the MATLAB Data Import and Export documentation for more information:

- "Mapping a Single Data Type"
- "Formatting the Mapped Data to an Array"
- "Mapping Multiple Data Types and Arrays"

Any of the following data types can be used when you specify a `Format` value. The default type is `uint8`.

Format String	Data Type Description
'int8'	Signed 8-bit integers
'int16'	Signed 16-bit integers
'int32'	Signed 32-bit integers
'int64'	Signed 64-bit integers
'uint8'	Unsigned 8-bit integers
'uint16'	Unsigned 16-bit integers
'uint32'	Unsigned 32-bit integers
'uint64'	Unsigned 64-bit integers
'single'	32-bit floating-point
'double'	64-bit floating-point

Tips

You can only map an existing file. You cannot create a new file and map that file to memory in one operation. Use the MATLAB file I/O functions to create the file before attempting to map it to memory.

Once `memmapfile` locates the file, MATLAB stores the absolute pathname for the file internally, and then uses this stored path to locate the file from that point on. This enables you to work in other directories outside your current work directory and retain access to the mapped file.

Once a `memmapfile` object has been constructed, you can change the value of any of its properties. Use the `objname.property` syntax in assigning the new value. To set a new offset value for memory map object `m`, type

```
m.Offset = 2048;
```

Property names are not case sensitive. For example, MATLAB considers `m.Offset` to be the same as `m.offset`.

Examples

To run the following examples, create a file in your current folder called `records.dat`. Example 2 expects that `records.dat` contains

5000 double-precision values. The following code shows one method to create this file:

```
randData = gallery('uniformdata', [5000, 1], 0, 'double');

fid = fopen('records.dat','w');
fwrite(fid, randData, 'double');
fclose(fid);
```

Example 1

To construct a map for `records.dat`, type the following:

```
m = memmapfile('records.dat');
```

MATLAB constructs an instance of the `memmapfile` class, assigns it to the variable `m`, and maps the entire `records.dat` file to memory, setting all properties of the object to their default values. In this example, the command maps the entire file as a sequence of unsigned 8-bit integers and gives the caller read-only access to its contents.

Example 2

To construct a map using nondefault values for the `Offset`, `Format`, and `Writable` properties, type the following, enclosing all property names in single quotation marks:

```
m = memmapfile('records.dat',         ...
               'Offset', 1024,        ...
               'Format', 'uint32',    ...
               'Writable', true);
```

Type the object name to see the current settings for all properties:

```
m

m =
    Filename: 'd:\matlab\records.dat'
    Writable: true
      Offset: 1024
```

```
Format: 'uint32'
Repeat: Inf
  Data: 9744x1 uint32 array
```

Example 3

Construct a `memmapfile` object for the entire file `records.dat` and set the `Format` property for that object to `uint64`. Any read or write operations made via the memory map will read and write the file contents as a sequence of unsigned 64-bit integers:

```
m = memmapfile('records.dat', 'Format', 'uint64');
```

Example 4

Construct a `memmapfile` object for a region of `records.dat` such that the contents of the region are handled by MATLAB as a 4-by-10-by-18 array of unsigned 32-bit integers, and can be referenced in the structure of the returned object using the field name `x`:

```
m = memmapfile('records.dat',    ...
               'Offset', 1024,   ...
               'Format', {'uint32' [4 10 18] 'x'});

A = m.Data(1).x;

whos A
  Name      Size                         Bytes  Class

  A         4x10x18                      2880   uint32 array
```

Example 5

Map a file to three different data types: `int16`, `uint32`, and `single`. The `int16` data is mapped as a 2-by-2 matrix that can be accessed using the field name `model`. The `uint32` data is a scalar value accessed as field `serialno`. The `single` data is a 1-by-3 matrix named `expenses`. Repeat the pattern 1000 times.

Each of the fields belongs to the 1000-by-1 structure array `m.Data`:

```
m = memmapfile('records.dat',                    ...
               'Format', {                       ...
                  'int16'  [2 2] 'model';        ...
                  'uint32' [1 1] 'serialno';     ...
                  'single' [1 3] 'expenses'},    ...
               'Repeat', 1000);
```

See Also

disp (memmapfile) | get (memmapfile)

Purpose

Display memory information

Syntax

```
memory
userview = memory
[userview systemview] = memory
```

Description

`memory` displays information showing how much memory is available and how much the MATLAB software is currently using. The information displayed at your computer screen includes the following items, each of which is described in a section below:

- "Maximum Possible Array" on page 1-3251
- "Memory Available for All Arrays" on page 1-3252
- "Memory Used By MATLAB" on page 1-3253
- "Total Physical Memory (RAM)" on page 1-3253

`userview = memory` returns user-focused information on memory use in structure `userview`. The information returned in `userview` includes the following items, each of which is described in a section below:

- "Maximum Possible Array" on page 1-3251
- "Memory Available for All Arrays" on page 1-3252
- "Memory Used By MATLAB" on page 1-3253

`[userview systemview] = memory` returns both user- and system-focused information on memory use in structures `userview` and `systemview`, respectively. The `userview` structure is described in the command syntax above. The information returned in `systemview` includes the following items, each of which is described in a section below:

- "Virtual Address Space" on page 1-3254
- "System Memory" on page 1-3254
- "Physical Memory" on page 1-3255

Output Arguments

Each of the sections below describes a value that is displayed or returned by the `memory` function.

Maximum Possible Array

Maximum Possible Array is the size of the largest contiguous free memory block. As such, it is an upper bound on the largest single array MATLAB can create at this time.

MATLAB derives this number from the smaller of the following two values:

- The largest contiguous memory block found in the MATLAB virtual address space
- The total available system memory

To see how many array elements this number represents, divide by the number of bytes in the array class. For example, for a `double` array, divide by 8. The actual number of elements MATLAB can create is always fewer than this number.

When you enter the `memory` command without assigning its output, MATLAB displays this information as a string. When you do assign the output, MATLAB returns the information in a structure field. See the table below.

Command	Returned in
`memory`	String labelled `Maximum possible array:`
`user = memory`	Structure field `user.MaxPossibleArrayBytes`

All values are double-precision and in units of bytes.

Footnotes

When you enter the `memory` command without specifying any outputs, MATLAB may also display one of the following footnotes. 32-bit systems show either the first or second footnote; 64-bit systems show only the second footnote:

`Limited by contiguous virtual address space available.`
: There is sufficient system memory to allow mapping of all virtual addresses in the largest available block of the MATLAB process. The maximum amount of total MATLAB virtual address space is either 2 GB or 3 GB, depending on whether the /3GB switch is in effect or not.

`Limited by System Memory (physical + swap file) available.`
: There is insufficient system memory to allow mapping of all virtual addresses in the largest available block of the MATLAB process.

Memory Available for All Arrays

Memory Available for All Arrays is the total amount of memory available to hold data. The amount of memory available is guaranteed to be at least as large as this field.

MATLAB derives this number from the smaller of the following two values:

- The total available MATLAB virtual address space
- The total available system memory

When you enter the `memory` command without assigning its output, MATLAB displays this information as a string. When you do assign the output, MATLAB returns the information in a structure field. See the table below.

Command	Returned in
`memory`	String labelled `Memory available for all arrays:`
`user = memory`	Structure field `user.MemAvailableAllArrays`

Footnotes

When you enter the `memory` command without specifying any outputs, MATLAB may also display one of the following footnotes. 32-bit systems show either the first or second footnote; 64-bit systems show only the latter footnote:

`Limited by virtual address space available.`
: There is sufficient system memory to allow mapping of all available virtual addresses in the MATLAB process virtual address space to system memory. The maximum amount of total MATLAB virtual address space is either 2 GB or 3 GB, depending on whether the /3GB switch is in effect or not.

`Limited by System Memory (physical + swap file) available.`
: There is insufficient system memory to allow mapping of all available virtual addresses in the MATLAB process.

Memory Used By MATLAB

Memory Used By MATLAB is the total amount of system memory reserved for the MATLAB process. It is the sum of the physical memory and potential swap file usage.

When you enter the `memory` command without assigning its output, MATLAB displays this information as a string. When you do assign the output, MATLAB returns the information in a structure field. See the table below.

Command	Returned in
`memory`	String labelled `Memory used by MATLAB:`
`user = memory`	Structure field `user.MemUsedMATLAB`

Total Physical Memory (RAM)

Physical Memory (RAM) is the total physical memory (or RAM) in the computer.

When you enter the `memory` command without assigning its output, MATLAB displays this information as a string. See the table below.

Command	Returned in
`memory`	String labelled `Physical Memory (RAM):`

Virtual Address Space

Virtual Address Space is the amount of available and total virtual memory for the MATLAB process. MATLAB returns the information in two fields of the return structure: `Available` and `Total`.

Command	Return Value	Returned in Structure Field
`[user,sys] = memory`	Available memory	`sys.VirtualAddressSpace.Available`
	Total memory	`sys.VirtualAddressSpace.Total`

You can monitor the difference:

```
VirtualAddressSpace.Total - VirtualAddressSpace.Available
```

as the Virtual Bytes counter in the WindowsPerformance program. (e.g., Windows XP Control Panel/Administrative Tool/Performance program).

System Memory

System Memory is the amount of available system memory on your computer system. This number includes the amount of available physical memory and the amount of available swap file space on the computer running MATLAB. MATLAB returns the information in the `SystemMemory` field of the return structure.

Command	Return Value	Returned in Structure Field
`[user,sys] = memory`	Available memory	`sys.SystemMemory`

This is the same as the difference:

```
limit - total (in bytes)
```

found in the Windows Task Manager: Performance/Commit Charge.

Physical Memory

Physical Memory is the available and total amounts of physical memory (RAM) on the computer running MATLAB. MATLAB returns the information in two fields of the return structure: `Available` and `Total`.

Command	Value	Returned in Structure Field
`[user,sys] = memory`	Available memory	`sys.PhysicalMemory.Available`
	Total memory	`sys.PhysicalMemory.Total`

Available physical memory is the same as:

```
Available (in bytes)
```

found in the Windows Task Manager: Performance/Physical Memory

The total physical memory is the same as

```
Total (in bytes)
```

found in the Windows Task Manager: Performance/Physical Memory

You can use the amount of available physical memory as a measure of how much data you can access quickly.

Tips

The `memory` function is currently available on Microsoft Windows systems only. Results vary, depending on the computer running MATLAB, the load on that computer, and what MATLAB is doing at the time.

Details on Memory Used By MATLAB

MATLAB computes the value for Memory Used By MATLAB by walking the MATLAB process memory structures and summing all the sections that have physical storage allocated in memory or in the paging file on disk.

Using the Windows Task Manager, you have for the MATLAB.exe image:

```
Mem Usage < MemUsedMATLAB < Mem Usage + VM Size (in bytes)
```

where both of the following are true:

- `Mem Usage` is the working set size in kilobytes.
- `VM Size` is the page file usage, or private bytes, in kilobytes.

The working set size is the portion of the MATLAB virtual address space that is *currently* resident in RAM and can be referenced without a memory page fault. The page file usage gives the portion of the MATLAB virtual address space that requires a backup that doesn't already exist. Another name for page file usage is *private bytes*. It includes all MATLAB variables and workspaces. Since some of the pages in the page file may also be part of the working set, this sum is an overestimate of `MemUseMATLAB`. Note that there are virtual pages in the MATLAB process space that already have a backup. For example, code loaded from EXEs and DLLs and memory-mapped files. If any part of those files is in memory when the memory builtin is called, that memory will be counted as part of `MemUsedMATLAB`.

Reserved Addresses

Reserved addresses are addresses sets aside in the process virtual address space for some specific future use. These reserved addresses reduce the size of `MemAvailableAllArrays` and can reduce the size of the current or future value of `MaxPossibleArrayBytes`.

Example 1 — Java Virtual Machine (JVM)

At MATLAB startup, part of the MATLAB virtual address space is reserved by the Java Virtual Machine (JVM) and cannot be used for storing MATLAB arrays.

Example 2 — Standard Windows Heap Manager

MATLAB, by default, uses the standard Windows heap manager except for a set of small preselected allocation sizes. One characteristic of this heap manager is that its behavior depends upon whether the requested allocation is less than or greater than the fixed number of 524,280 bytes.

For, example, if you create a sequence of MATLAB arrays, each less then 524,280 bytes, and then clear them all, the MemUsedMATLAB value before and after shows little change, and the MemAvailableAllArrays value is now smaller by the total space allocated.

The result is that, instead of globally freeing the extra memory, the memory becomes reserved. It can *only* be reused for arrays less than 524,280 bytes. You cannot reclaim this memory for a larger array except by restarting MATLAB.

Examples

Display memory statistics on a 32-bit Windows system:

```
memory

Maximum possible array:           677 MB (7.101e+008 bytes) *
Memory available for all arrays: 1601 MB (1.679e+009 bytes) **
Memory used by MATLAB:            446 MB (4.681e+008 bytes)
Physical Memory (RAM):           3327 MB (3.489e+009 bytes)

  *  Limited by contiguous virtual address space available.
  ** Limited by virtual address space available.
```

Return in the structure userview, information on the largest array MATLAB can create at this time, how much memory is available to hold data, and the amount of memory currently being used by your MATLAB process:

```
userview = memory

userview =
   MaxPossibleArrayBytes: 710127616
   MemAvailableAllArrays: 1.6792e+009
            MemUsedMATLAB: 468127744
```

Assign the output to two structures, user and sys, to obtain the information shown here:

```
[user sys] = memory;
```

```
%  --- Largest array MATLAB can create ---
user.MaxPossibleArrayBytes
ans =
   710127616

%  --- Memory available for data ---
user.MemAvailableAllArrays
ans =
   1.6797e+009

%  --- Memory used by MATLAB process ---
user.MemUsedMATLAB
ans =
   467603456

%  --- Virtual memory for MATLAB process ---
sys.VirtualAddressSpace
ans =
   Available: 1.6797e+009
       Total: 2.1474e+009

%  --- Physical memory and paging file ---
sys.SystemMemory
ans =
   Available: 4.4775e+009

%   --- Computer's physical memory ---
sys.PhysicalMemory
ans =
   Available: 2.3941e+009
       Total: 3.4889e+009
```

See Also

`clear` | `pack` | `whos` | `inmem` | `save` | `load` | `mlock` | `munlock`

Tutorials

- "Memory Allocation"

- "Strategies for Efficient Use of Memory"
- "Resolving "Out of Memory" Errors"

Purpose

Generate menu of choices for user input

Syntax

```
choice = menu('mtitle','opt1','opt2',...,'optn')
choice = menu('mtitle',options)
```

Description

`choice = menu('mtitle','opt1','opt2',...,'optn')` displays the menu whose title is in the string variable `'mtitle'` and whose choices are string variables `'opt1'`, `'opt2'`, and so on. The menu opens in a modal dialog box. `menu` returns the number of the selected menu item, or 0 if the user clicks the close button on the window.

`choice = menu('mtitle',options)` , where options is a 1-by-N cell array of strings containing the menu choices.

If the user's terminal provides a graphics capability, `menu` displays the menu items as push buttons in a figure window (Example 1). Otherwise. they will be given as a numbered list in the Command Window (Example 2).

Tips

To call `menu` from a uicontrol or other ui object, set that object's `Interruptible` property to `'on'`. For more information, see Uicontrol Properties.

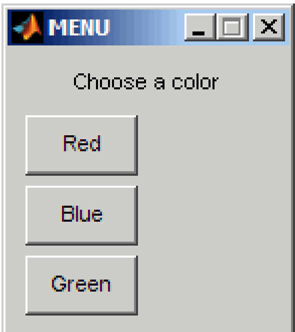
Examples

Example 1

On a system with a display, `menu` displays choices as buttons in a dialog box:

```
choice = menu('Choose a color','Red','Blue','Green')
```

displays the following dialog box.



The number entered by the user in response to the prompt is returned as `choice` (i.e., `choice = 2` implies that the user selected `Blue`).

After input is accepted, the dialog box closes, returning the output in `choice`. You can use `choice` to control the color of a graph:

```
t = 0:.1:60;
s = sin(t);
color = ['r','g','b']
plot(t,s,color(choice))
```

Example 2

On a system without a display, `menu` displays choices in the Command Window:

```
choice = menu('Choose a color','Red','Blue','Green')
```

displays the following text.

```
----- Choose a color -----
1) Red
2) Blue
3) Green
Select a menu number:
```

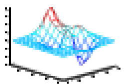
See Also

`guide` | `input` | `uicontrol` | `uimenu`

mesh

Purpose

Mesh plot



Syntax

```
mesh(X,Y,Z)
mesh(Z)
mesh(...,C)
mesh(...,'PropertyName',PropertyValue,...)
mesh(axes_handles,...)
h = mesh(...)
```

Description

`mesh(X,Y,Z)` draws a wireframe mesh with color determined by `Z`, so color is proportional to surface height. If `X` and `Y` are vectors, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(Z)`. In this case, ($X(j)$, $Y(i)$, $Z(i,j)$) are the intersections of the wireframe grid lines; `X` and `Y` correspond to the columns and rows of `Z`, respectively. If `X` and `Y` are matrices, ($X(i,j)$, $Y(i,j)$, $Z(i,j)$) are the intersections of the wireframe grid lines.

`mesh(Z)` draws a wireframe mesh using `X = 1:n` and `Y = 1:m`, where `[m,n] = size(Z)`. The height, `Z`, is a single-valued function defined over a rectangular grid. Color is proportional to surface height.

`mesh(...,C)` draws a wireframe mesh with color determined by matrix `C`. MATLAB performs a linear transformation on the data in `C` to obtain colors from the current colormap. If `X`, `Y`, and `Z` are matrices, they must be the same size as `C`.

`mesh(...,'PropertyName',PropertyValue,...)` sets the value of the specified surface property. Multiple property values can be set with a single statement.

`mesh(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = mesh(...)` returns a handle to a Surfaceplot graphics object.

Tips

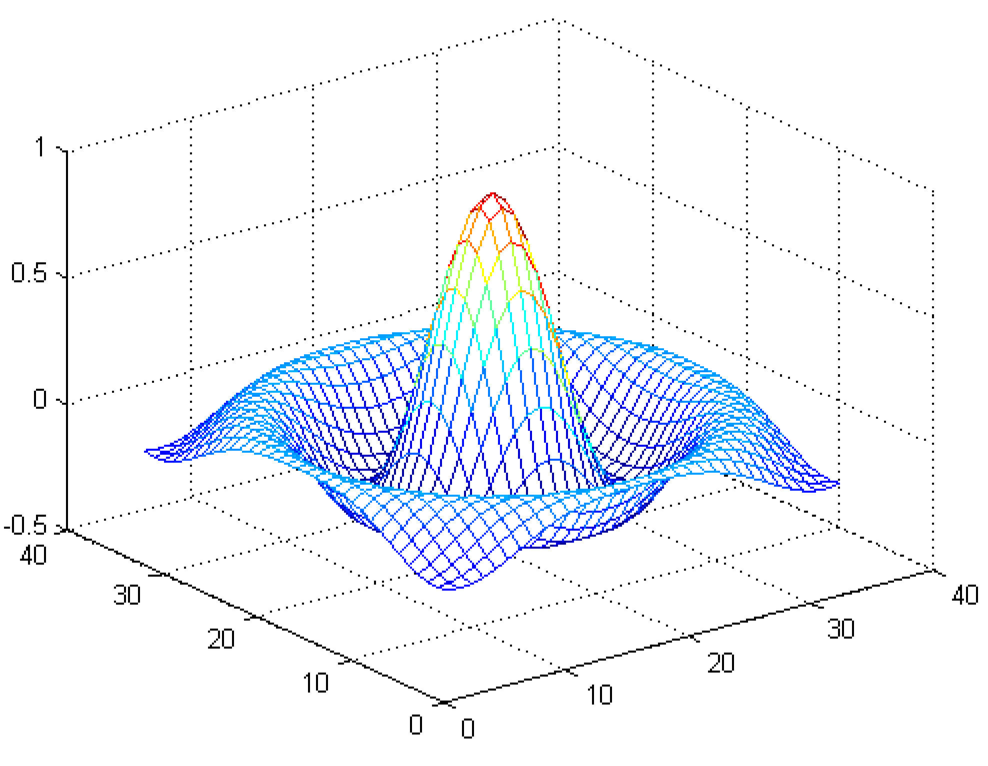
mesh does not accept complex inputs.

A mesh is drawn as a Surfaceplot graphics object with the viewpoint specified by view(3). The face color is the same as the background color (to simulate a wireframe with hidden-surface elimination), or none when drawing a standard see-through wireframe. The current colormap determines the edge color. The hidden command controls the simulation of hidden-surface elimination in the mesh, and the shading command controls the shading model.

Examples

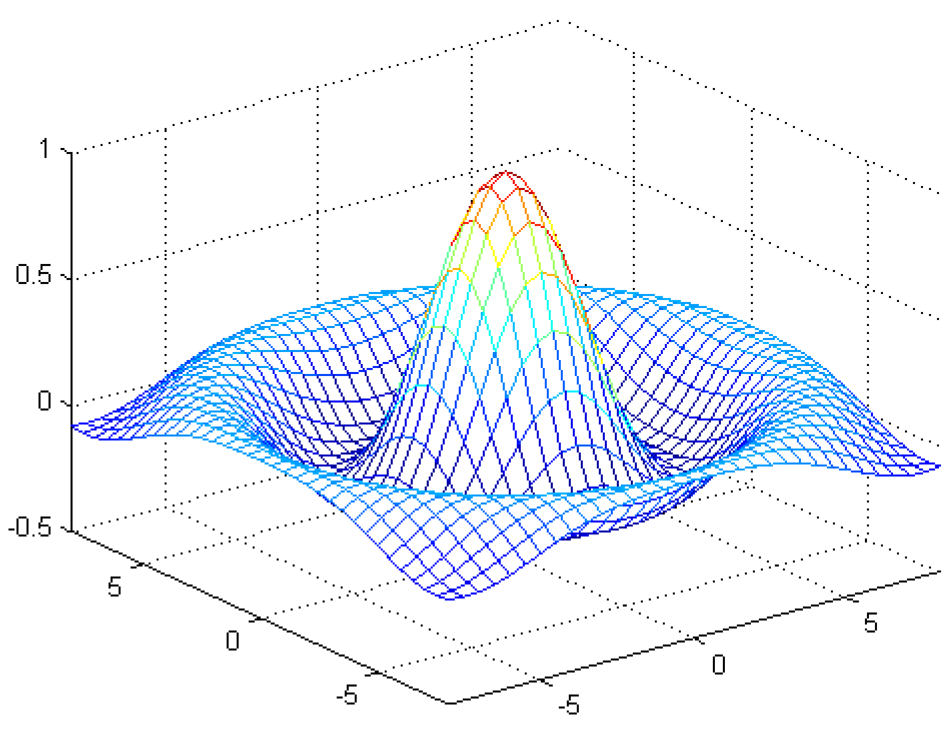
Evaluate sin(r)/r, or the *sinc* function

```
figure;
[X,Y] = meshgrid(-8:.5:8);
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R)./R;
mesh(Z);
```



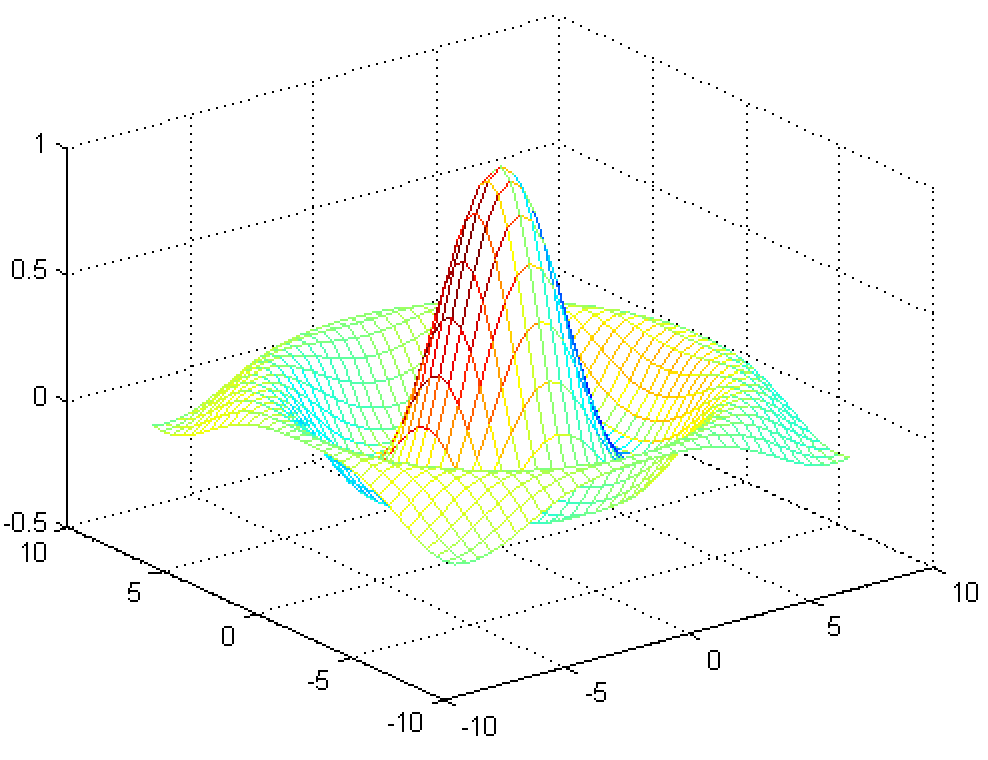
Displaying the *sinc* function between -8 and 8 on a 2-D grid

```
figure;
mesh(X,Y,Z);
axis([-8 8 -8 8 -0.5 1]);
```



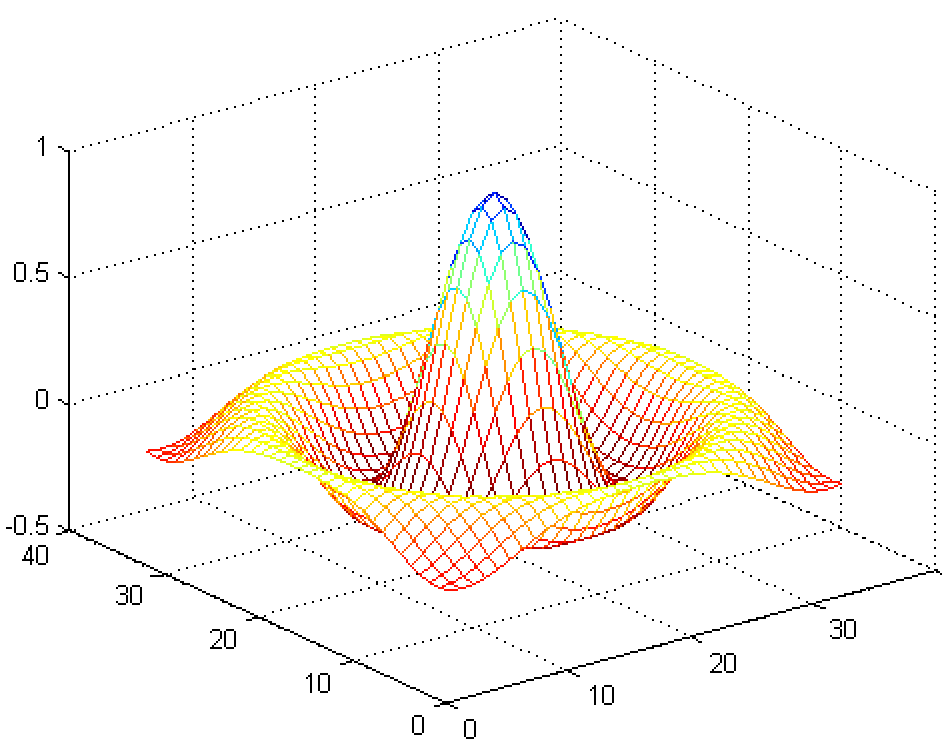
Specifying color matrix for the data in the mesh function

```
C = gradient(Z);
figure;
mesh(X,Y,Z,C);
```



Specifying the property name-value pairs in the argument list.

```
C = del2(Z);
h = axes;
mesh(h,Z,C,'FaceLighting','gouraud','LineWidth',0.3);
```



See Also

`meshc` | `meshz` | `hidden` | `griddata` | `scatteredInterpolant` | `meshgrid` | `surface` | `surf` | `waterfall` | `axis` | `colormap` | `hold` | `shading` | `view`

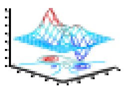
How To

- "Representing Data as a Surface"

meshc

Purpose

Plot a contour graph under mesh graph



Syntax

```
meshc(X,Y,Z)
meshc(Z)
meshc(...,C)
meshc(axes_handles,...)
h = meshc(...)
```

Description

`meshc(X,Y,Z)` draws a wireframe mesh and a contour plot under it with color determined by `Z`, so color is proportional to surface height. If `X` and `Y` are vectors, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(Z)`. In this case, $(X(j), Y(i), Z(i,j))$ are the intersections of the wireframe grid lines; `X` and `Y` correspond to the columns and rows of `Z`, respectively. If `X` and `Y` are matrices, $(X(i,j), Y(i,j), Z(i,j))$ are the intersections of the wireframe grid lines.

`meshc(Z)` draws a contour plot under wireframe mesh using `X = 1:n` and `Y = 1:m`, where `[m,n] = size(Z)`. The height, `Z`, is a single-valued function defined over a rectangular grid. Color is proportional to surface height.

`meshc(...,C)` draws a `meshc` graph with color determined by matrix `C`. MATLAB performs a linear transformation on the data in `C` to obtain colors from the current colormap. If `X`, `Y`, and `Z` are matrices, they must be the same size as `C`.

`meshc(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = meshc(...)` returns a handle to a Surfaceplot graphics object.

Tips

`meshc` does not accept complex inputs.

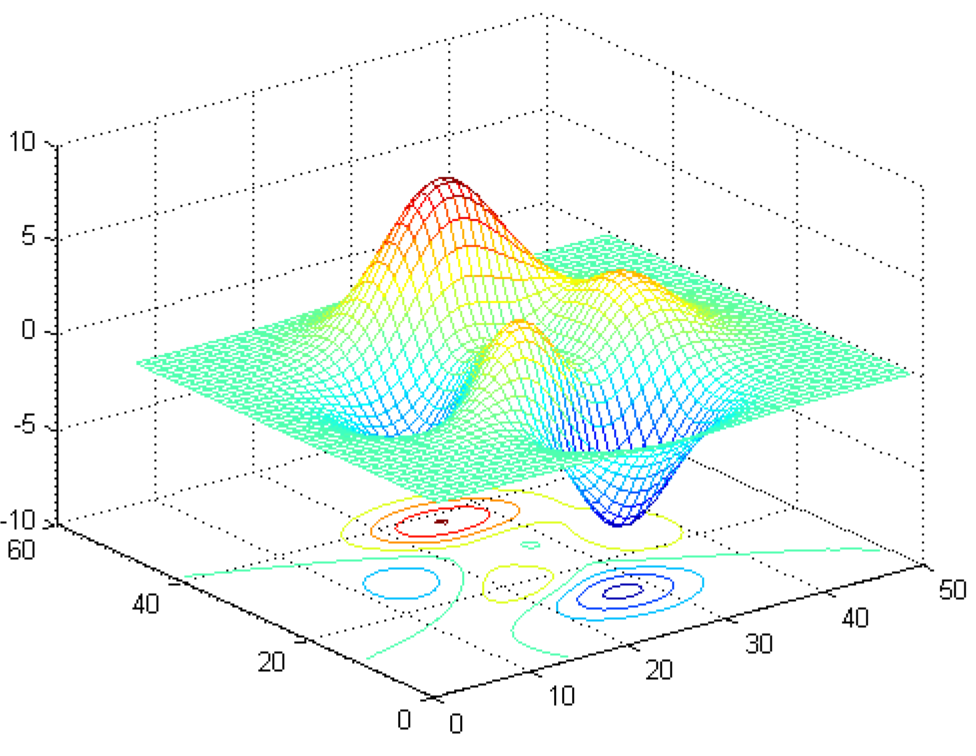
A mesh is drawn as a `Surfaceplot` graphics object with the viewpoint specified by `view(3)`. The face color is the same as the background

color (to simulate a wireframe with hidden-surface elimination), or `none` when drawing a standard see-through wireframe. The current colormap determines the edge color. The `hidden` command controls the simulation of hidden-surface elimination in the mesh, and the `shading` command controls the shading model.

Examples

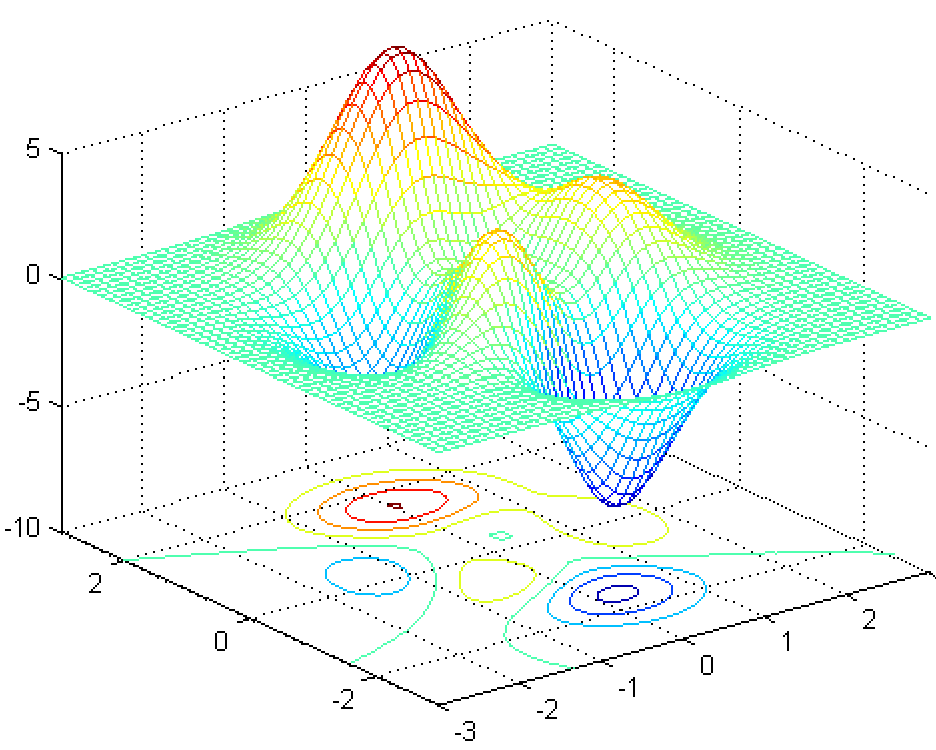
Produce a combination mesh and contour plot of the `peaks` surface with `meschc`:

```
figure
[X,Y] = meshgrid(-3:.125:3);
Z = peaks(X,Y);
meshc(Z);
```



Specifying axes limits for the `meshc` function.

```
figure;
meshc(X,Y,Z);
axis([-3 3 -3 3 -10 5]);
```



Algorithms

The range of `X`, `Y`, and `Z`, or the current settings of the axes `XLimMode`, `YLimMode`, and `ZLimMode` properties, determine the axis limits. `axis` sets these properties.

The range of `C`, or the current settings of the axes `CLim` and `CLimMode` properties (also set by the `caxis` function), determine the color scaling. Use the scaled color values are used as indices into the current colormap.

The mesh rendering functions produce color values by mapping the *z* data values (or an explicit color array) onto the current colormap. The

MATLAB default behavior is to compute the color limits automatically using the minimum and maximum data values (also set using `caxis auto`). The minimum data value maps to the first color value in the colormap and the maximum data value maps to the last color value in the colormap. MATLAB performs a linear transformation on the intermediate values to map them to the current colormap.

`meshc` calls `mesh`, turns `hold` on, and then calls `contour` and positions the contour on the *x*-*y* plane. For additional control over the appearance of the contours, issue these commands directly. You can combine other types of graphs in this manner, for example `surf` and `pcolor` plots.

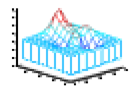
`meshc` assumes that `X` and `Y` are monotonically increasing. If `X` or `Y` is irregularly spaced, `contour3` calculates contours using a regularly spaced contour grid, and then it transforms the data to `X` or `Y`.

See Also

`mesh` | `meshz` | `contour` | `hidden` | `meshgrid` | `surface` | `surf` | `surfc` | `surfl` | `waterfall` | `axis` | `caxis` | `colormap` | `hold` | `shading` | `view`

Purpose

Plot a curtain around mesh plot



Syntax

```
meshz(X,Y,Z)
meshz(Z)
meshz(...,C)
meshz(axes_handles,...)
h = meshz(...)
```

Description

`meshz(X,Y,Z)` draws a curtain around the wireframe mesh with color determined by `Z`, so color is proportional to surface height. If `X` and `Y` are vectors, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(Z)`. In this case, $(X(j), Y(i), Z(i,j))$ are the intersections of the wireframe grid lines; `X` and `Y` correspond to the columns and rows of `Z`, respectively. If `X` and `Y` are matrices, $(X(i,j), Y(i,j), Z(i,j))$ are the intersections of the wireframe grid lines.

`meshz(Z)` draws a curtain around the wireframe mesh using `X = 1:n` and `Y = 1:m`, where `[m,n] = size(Z)`. The height, `Z`, is a single-valued function defined over a rectangular grid. Color is proportional to surface height.

`meshz(...,C)` draws a `meshz` graph with color determined by matrix `C`. MATLAB performs a linear transformation on the data in `C` to obtain colors from the current colormap. If `X`, `Y`, and `Z` are matrices, they must be the same size as `C`.

`meshz(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = meshz(...)` returns a handle to a Surfaceplot graphics object.

Tips

`meshz` does not accept complex inputs.

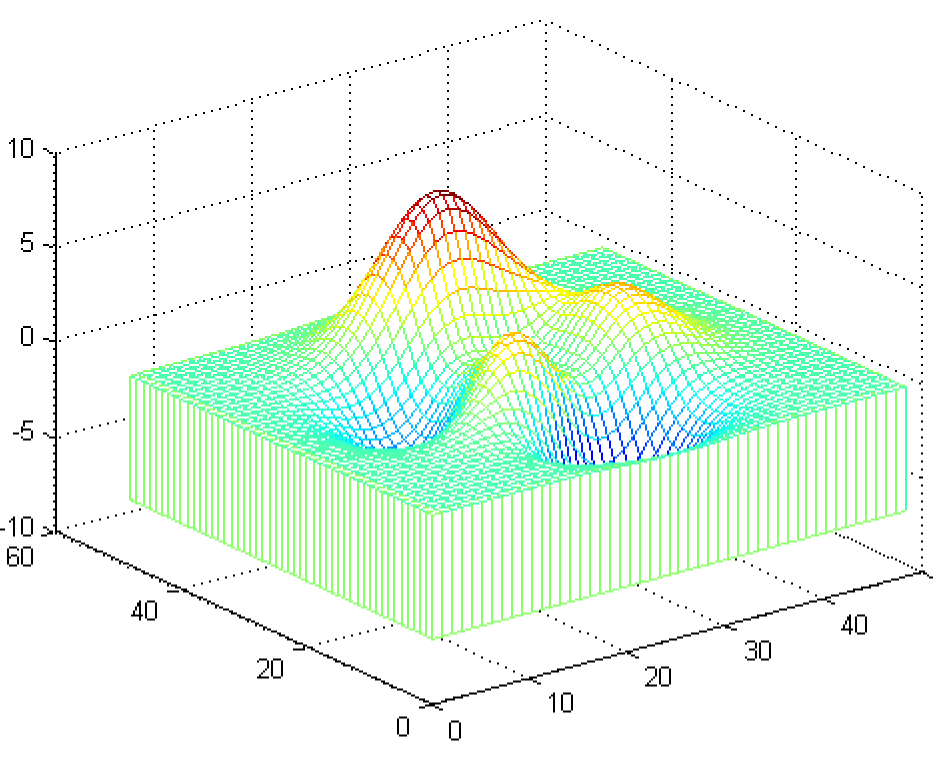
A mesh is drawn as a `Surfaceplot` graphics object with the viewpoint specified by `view(3)`. The face color is the same as the background

color (to simulate a wireframe with hidden-surface elimination), or `none` when drawing a standard see-through wireframe. The current colormap determines the edge color. The `hidden` command controls the simulation of hidden-surface elimination in the mesh, and the `shading` command controls the shading model.

Examples

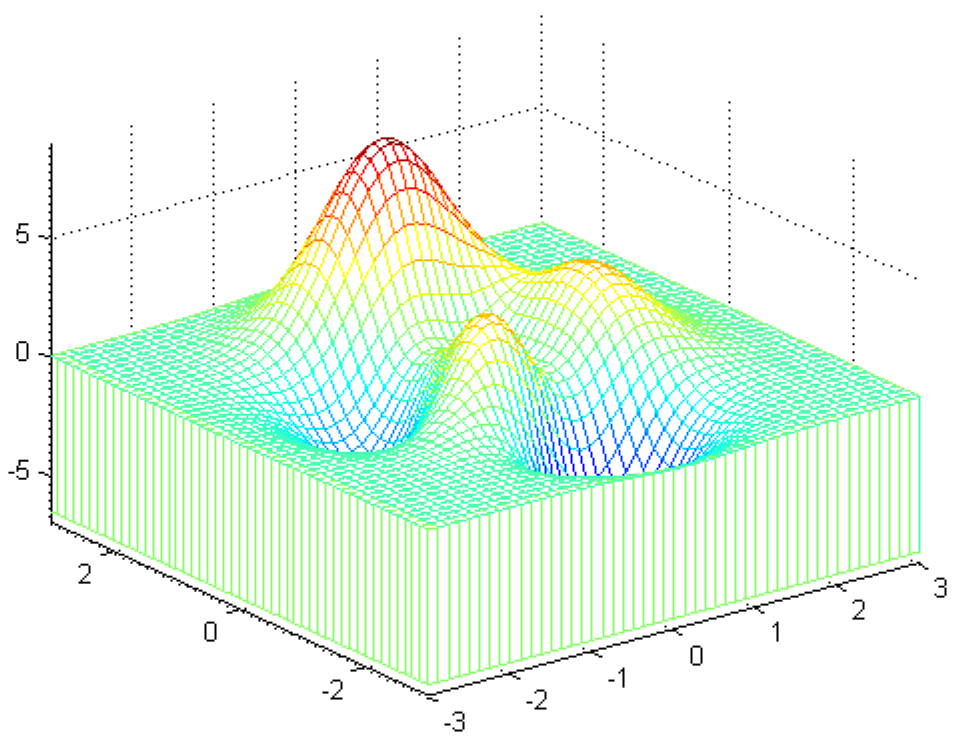
Generate a curtain plot for the `peaks` function using `meshz`

```
figure
[X,Y] = meshgrid(-3:.125:3);
Z = peaks(X,Y);
meshz(Z);
```



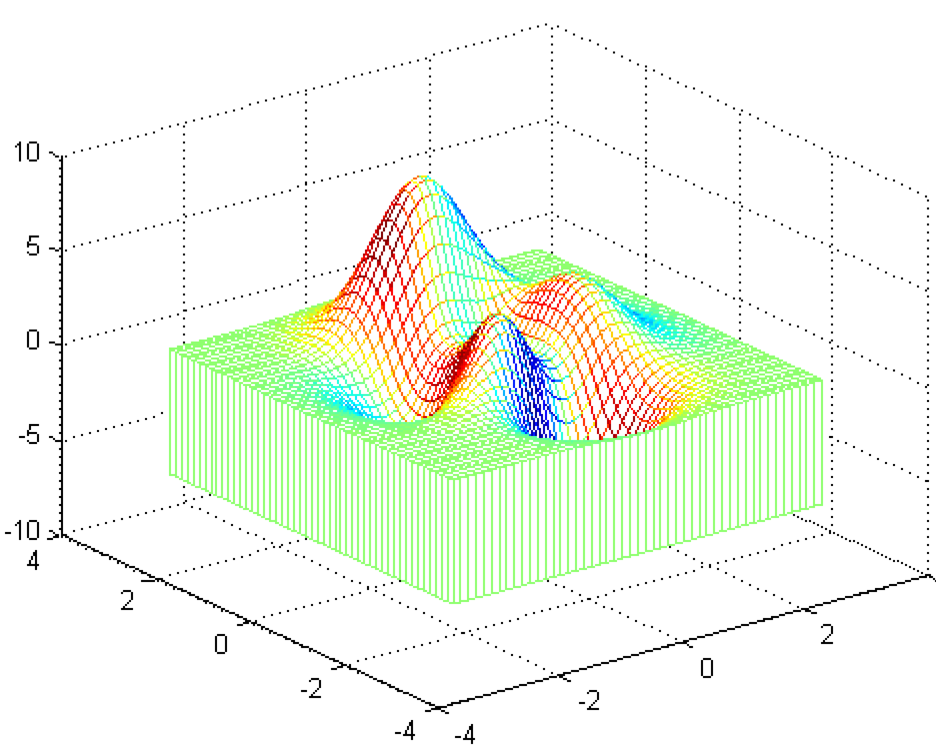
Specifying axes limits for the graph

```
figure;
meshz(X,Y,Z);
axis([-3 3 -3 3 -7 9]);
```



Specifying a color matrix

```
C = gradient(Z);
figure;
meshz(X,Y,Z,C);
```



See Also

mesh | meshc | contour | hidden | meshgrid | surface | surf | surfc | surfl | waterfall | axis | caxis | colormap | hold | shading | view

meshgrid

Purpose

Rectangular grid in 2-D and 3-D space

Syntax

```
[X,Y] = meshgrid(xgv,ygv)
[X,Y,Z] = meshgrid(xgv,ygv,zgv)
[X,Y] = meshgrid(gv)
[X,Y,Z] = meshgrid(gv)
```

Description

`[X,Y] = meshgrid(xgv,ygv)` replicates the grid vectors `xgv` and `ygv` to produce a full grid. This grid is represented by the output coordinate arrays `X` and `Y`. The output coordinate arrays `X` and `Y` contain copies of the grid vectors `xgv` and `ygv` respectively. The sizes of the output arrays are determined by the length of the grid vectors. For grid vectors `xgv` and `ygv` of length `M` and `N` respectively, `X` and `Y` will have `N` rows and `M` columns.

`[X,Y,Z] = meshgrid(xgv,ygv,zgv)` produces three-dimensional coordinate arrays. The output coordinate arrays `X`, `Y`, and `Z` contain copies of the grid vectors `xgv`, `ygv`, and `zgv` respectively. The sizes of the output arrays are determined by the length of the grid vectors. For grid vectors `xgv`, `ygv`, and `zgv` of length `M`, `N`, and `P` respectively, `X`, `Y`, and `Z` will have `N` rows, `M` columns, and `P` pages.

`[X,Y] = meshgrid(gv)` is the same as `[X,Y] = meshgrid(gv,gv)`. In other words, you can reuse the same grid vector in each respective dimension. The dimensionality of the output arrays is determined by the number of output arguments.

`[X,Y,Z] = meshgrid(gv)` is the same as `[X,Y,Z] = meshgrid(gv,gv,gv)`. Again, the dimensionality of the output arrays is determined by the number of output arguments.

The output coordinate arrays are typically used to evaluate functions of two or three variables. They are also frequently used to create surface and volumetric plots.

Input Arguments

xgv,ygv,zgv

Grid vectors specifying a series of grid point coordinates in the `x`, `y` and `z` directions, respectively.

gv

Generic grid vector specifying a series of point coordinates.

Output Arguments

X,Y,Z

Output arrays that specify the full grid.

Tips

The `meshgrid` function is similar to `ndgrid`, however `meshgrid` is restricted to 2-D and 3-D while `ndgrid` supports 1-D to N-D. The coordinates output by each function are the same, but the shape of the output arrays in the first two dimensions are different. For grid vectors `x1gv`, `x2gv` and `x3gv` of length `M`, `N` and `P` respectively, `meshgrid(x1gv, x2gv)` will output arrays of size `N`-by-`M` while `ndgrid(x1gv, x2gv)` outputs arrays of size `M`-by-`N`. Similarly, `meshgrid(x1gv, x2gv, x3gv)` will output arrays of size `N`-by-`M`-by-`P` while `ndgrid(x1gv, x2gv, x3gv)` outputs arrays of size `M`-by-`N`-by-`P`. See "Grid Representation" in the MATLAB Mathematics documentation for more information.

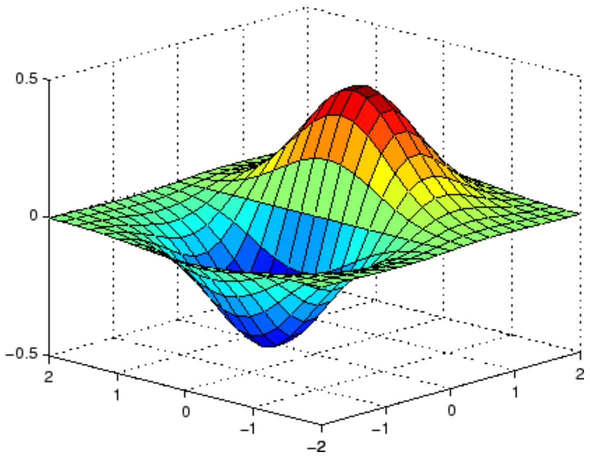
Examples

Create a full grid from two monotonically increasing grid vectors:

```
[X,Y] = meshgrid(1:3,10:14)
X =
     1     2     3
     1     2     3
     1     2     3
     1     2     3
     1     2     3
Y =
    10    10    10
    11    11    11
    12    12    12
    13    13    13
    14    14    14
```

Use `meshgrid` to create a 3-D surface plot of a function:

```
[X,Y] = meshgrid(-2:.2:2, -2:.2:2);
Z = X .* exp(-X.^2 - Y.^2);
surf(X,Y,Z)
```



See Also

`griddedInterpolant` | `mesh` | `ndgrid` | `surf`

How To

- “Interpolating Gridded Data”

Purpose

`meta.class` class describes MATLAB classes

Description

Instances of the `meta.class` class contains information about MATLAB classes. The read/write properties of the `meta.class` class correspond to class attributes and are set only from within class definitions on the `classdef` line. You can query the read–only properties of the `meta.class` object to obtain information that is specified syntactically by the class (for example, to obtain the name of the class).

You cannot instantiate a `meta.class` object directly. You can construct a `meta.class` object from an instance of a class or using the class name:

- `metaclass` — returns a `meta.class` object representing the object passed as an argument.
- `?`*`ClassName`* — returns a `meta.class` object representing the named class.
- `fromName` — static method returns a `meta.class` object representing the named class.

For example, the `metaclass` function returns the `meta.class` object representing `MyClass`.

```
ob = MyClass;
obmeta = metaclass(ob);
obmeta.Name
ans =
MyClass
```

You can use the class name to obtain the `meta.class` object:

```
obmeta = ?MyClass;
```

You can also use the `fromName` static method:

```
obmeta = meta.class.fromName('MyClass');
```

Properties

Property	Purpose
`Abstract` attribute, default = `false`	If `true`, this class is an abstract class (cannot be instantiated). See "Defining Abstract Classes" for more information.
`AllowedSubclasses` attribute, write only from `classdef` statement	List classes that can subclass this class. Specify subclasses as `meta.class` objects in the form: • A single `meta.class` object • A cell array of `meta.class` objects Specify `meta.class` objects using the `?ClassName` syntax only.
`ConstructOnLoad` attribute, default = `false`	If `true`, MATLAB calls the class constructor automatically when loading an object from a MAT-file. Therefore, the construction must be implemented so that calling it with no arguments does not produce an error. See "Tips for Saving and Loading"
`ContainingPackage` read only	A `meta.package` object describing the package within which this class is contained, or an empty object if this class is not in a package. See "Packages Create Namespaces".
`Description` read only	Currently not used
`DetailedDescription` read only	Currently not used

Property	Purpose
`EventList` read only	An array of `meta.event` objects describing each event defined by this class, including all inherited events. See "Events".
`Events` read onlyUse `EventList` instead	A cell array of `meta.event` objects describing each event defined by this class, including all inherited events.
`EnumerationMemberList`	An array of `meta.EnumeratedValue` objects describing the member names defined by an enumeration class. See "Enumerations" for more information on enumeration classes.
`EnumeratedValues` read only Use `EnumeratedMemberList` instead	A cell array of `meta.EnumeratedValue` objects describing the member names defined by an enumeration class. See "Enumerations" for more information on enumeration classes.
`Hidden` attribute, default = `false`	If set to `true`, the class does not appear in the output of MATLAB commands or tools that display class names.
`InferiorClasses` attribute, default = `{}`	A cell array of `meta.class` objects defining the precedence of classes represented by the list as inferior to this class. See "Class Precedence"
`MethodList` read only	An array of `meta.method` objects describing each method defined by this class, including all inherited public and protected methods. See "How to Use Methods".

Property	Purpose
`Methods` read only Use `MethodList` instead	A cell array of `meta.method` objects describing each method defined by this class, including all inherited public and protected methods.
`Name` read only	Name of the class associated with this `meta.class` object (`char` array)
`PropertyList` read only	An array of `meta.property` objects describing each property defined by this class, including all inherited public and protected properties. See "Properties".
`Properties` read only Use `PropertyList` instead	A cell array of `meta.property` objects describing each property defined by this class, including all inherited public and protected properties. See "Properties".
`Sealed` attribute, default = `false`	If `true`, the class cannot be subclassed.
`SuperClassList` read only	An array of `meta.class` objects describing each direct superclass from which this class is derived. See "Creating Subclasses — Syntax and Techniques".
`SuperClasses` read only Use `SuperClassList` instead	A cell array of `meta.class` objects describing each direct superclass from which this class is derived.

Methods

Method	Purpose
`fromName`	Returns the `meta.class` object associated with the specified class name.
`tf = eq(Cls)`	Equality function (`a == b`). Use to test if two variables refer to equal classes (classes that contain exactly the same list of elements).
`tf = ne(Cls)`	Not equal function (`a ~= b`). Use to test if two variables refer to different meta–classes.
`tf = lt(ClsA,ClsB)`	Less than function (`ClsA < ClsB`). Use to determine if `ClsA` is a strict subclass of `ClsB` (i.e., a strict subclass means `ClsX < ClsX` is `false`).
`tf = le(ClsA,ClsB)`	Less than or equal to function (`ClsA <= ClsB`). Use to determine if `ClsA` is a subclass of `ClsB`.
`tf = gt(ClsA,ClsB)`	Greater than function (`ClsB > ClsA`). Use to determine if `ClsB` is a strict superclass of `ClsA` (i.e., a strict superclass means `ClsX > ClsX` is `false`).
`tf = ge(ClsA,ClsB)`	Greater than or equal to function (`ClsB >= ClsA`). Use to determine if `ClsB` is a superclass of `ClsA`.

See Also

`fromName` | `meta.property` | `meta.method` | `meta.event` | `meta.package`

How To

- "Getting Information About Classes and Objects"

meta.class.fromName

Purpose

Return `meta.class` object associated with named class

Syntax

```
mcls = meta.class.fromName('ClassName')
```

Description

`mcls = meta.class.fromName('ClassName')` is a static method that returns the `meta.class` object associated with the class *`ClassName`*. Note that you can also use the `?` operator to obtain the `meta.class` object for a class name:

```
mcls = ?ClassName;
```

The equivalent call to `meta.class.fromName` is:

```
mcls = meta.class.fromName('ClassName');
```

Use `meta.class.fromName` when using a `char` variable for the class name:

```
function mcls = getMetaClass(clname)
   % Do error checking
   mcls = meta.class.fromName(clname);
   ...
end
```

See Also

`meta.class`

Purpose

`meta.DynamicProperty` class describes dynamic property of MATLAB object

Description

The `meta.DynamicProperty` class contains descriptive information about dynamic properties that you have added to an instance of a MATLAB classes. The MATLAB class must be a subclass of `dynamicprops`. The properties of the `meta.DynamicProperty` class correspond to property attributes that you specify from within class definitions. Dynamic properties are not defined in `classdef` blocks, but you can set their attributes by setting the `meta.DynamicProperty` object properties.

You add a dynamic property to an object using the `addprop` method of the `dynamicprops` class. The `addprop` method returns a `meta.DynamicProperty` instance representing the new dynamic property. You can modify the properties of the `meta.DynamicProperty` object to set the attributes of the dynamic property or to add set and get access methods, which would be defined in the `classdef` for regular properties.

You cannot instantiate the `meta.DynamicProperty` class. You must use `addprop` to obtain a `meta.DynamicProperty` object.

To remove the dynamic property, call the `delete` handle class method on the `meta.DynamicProperty` object.

Obtain a `meta.DynamicProperty` object from the `addprops` method, which returns an array of `meta.DynamicProperty` objects, one for each dynamic property.

See "Dynamic Properties — Adding Properties to an Instance" for more information.

Properties

Property	Purpose
`Name`	Name of the property.
`Description`	Currently not used

Property	Purpose
DetailedDescription	Currently not used
AbortSet	If true, and this property belongs to a handle class, then MATLAB does not set the property value if the new value is the same as the current value. This approach prevents the triggering of property PreSet and PostSet events.
Abstract attribute, default = false	If true, the property has no implementation, but a concrete subclass must redefine this property without Abstract being set to true. • Abstract properties cannot define set or get access methods. See "Property Access Methods" • Abstract properties cannot define initial values. "Assigning a Default Value" • All subclasses must specify the same values as the superclass for the property SetAccess and GetAccess attributes. • Abstract=true should be used with the class attribute Sealed=false (the default).

<table>
<tr><th>Property</th><th>Purpose</th></tr>
<tr><td><code>Access</code></td><td><code>public</code> – unrestricted access

<code>protected</code> – access from class or subclasses

<code>private</code> – access by class members only (not subclasses)

List of classes that have get and set access to this property. Specify classes as <code>meta.class</code> objects in the form:

• A single <code>meta.class</code> object
• A cell array of <code>meta.class</code> objects. An empty cell array, <code>{}</code>, is the same as <code>private</code> access.

Use <code>Access</code> to set both <code>SetAccess</code> and <code>GetAccess</code> to the same value. Query the values of <code>SetAccess</code> and <code>GetAccess</code> directly (not <code>Access</code>).</td></tr>
<tr><td><code>Constant</code> attribute, default = <code>false</code></td><td>Set to <code>true</code> if you want only one value for this property in all instances of the class.

• Subclasses inherit constant properties, but cannot change them.
• <code>Constant</code> properties cannot be <code>Dependent</code>
• <code>SetAccess</code> is ignored.

See “Properties with Constant Values”</td></tr>
<tr><td><code>DefaultValue</code></td><td>Querying this property returns an error because dynamic properties cannot define default values.</td></tr>
</table>

Property	Purpose
`DefiningClass`	The `meta.class` object representing the class that defines this property.
`GetAccess` attribute, default = `public`	`public` – unrestricted access `protected` – access from class or subclasses `private` – access by class members only
`SetAccess` attribute, default = `public`	`public` – unrestricted access `protected` – access from class or subclasses `private` – access by class members only
`Dependent` attribute, default = `false`	If `false`, property value is stored in object. If `true`, property value is not stored in object and the set and get functions cannot access the property by indexing into the object using the property name. See "Property Get Methods"
`Transient` attribute, default = `false`	If `true`, property value is not saved when object is saved to a file. See "Understanding the Save and Load Process" for more about saving objects.
`Hidden` attribute, default = `false`	Determines whether the property should be shown in a property list (e.g., Property Inspector, call to `properties`, etc.).
`GetObservable` attribute, default = `false`	If `true`, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are queried. See "Property-Set and Query Events"

Property	Purpose
SetObservable attribute, default = false	If true, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are modified. See "Property-Set and Query Events"
GetMethod	Function handle of the get method associated with this property. Empty if there is no get method specified. See "Get Method Syntax"
SetMethod	Function handle of the set method associated with this property. Empty if there is no set method specified. See "Property Set Methods"
HasDefault	Always false. Dynamic properties cannot define default values.

Events

See "Listen for Changes to Property Values" for information on using property events.

Event Name	Purpose
PreGet	Event occurs just before property is queried.
PostGet	Event occurs just after property has been queried
PreSet	Event occurs just before this property is modified
PostSet	Event occurs just after this property has been modified
ObjectBeingDestroyed	Inherited from handle

See Also

addprop | handle

meta.EnumeratedValue

Purpose

Describes enumeration members of MATLAB class

Description

The meta.EnumeratedValue class contains information about enumeration members defined by MATLAB classes. The properties of a meta.EnumeratedValue object correspond the to attributes of the enumeration member being described.

All meta.EnumeratedValue properties are read-only. Query the meta.EnumeratedValue object to obtain information about the enumeration member it describes.

Obtain a meta.EnumeratedValue object from the EnumerationMemberList property of the meta.class object. EnumerationMemberList is an array of Meta.EnumeratedValue instances, one per enumeration member.

The meta.EnumeratedValue class is a subclass of the handle class.

Example

To access the meta.EnumeratedValue objects for a class, first create a meta.class object for that class. For example, give the following OnOff class definition:

```
classdef OnOff < logical
   enumeration
      On (true)
      Off (false)
   end
end
```

Obtain a meta.EnumeratedValue object from the EnumerationMemberList property of the meta.class object:

```
% Obtain the meta.class instance for the OnOff class
mc = ?OnOff;
% Get the array of EnumerateValue objects
enumList = mc.EnumerationMemberList;
% Access the Name property of the first object in the array
enumList(1).Name =
```

```
ans =
On
```

Properties

Property	Purpose
`Name` read only	Name of the enumeration member associated with this `meta.EnumeratedValue` object
`Description` read only	This property is not used
`DetailedDescription` read only	This property is not used

Methods

See the `handle` superclass for inherited methods.

Events

See the `handle` superclass for inherited events.

See Also

`meta.class` | `meta.property` | `meta.method` | `meta.event`

How To

- "Working with Enumerations"
- "Getting Information About Classes and Objects"

Purpose

`meta.event` class describes MATLAB class events

Description

The `meta.event` class provides information about MATLAB class events. The read/write properties of the `meta.event` class correspond to event attributes and are specified only from within class definitions.

You can query the read-only properties of the `meta.event` object to obtain information that is specified syntactically by the class (for example, to obtain the name of the class defining the event).

You cannot instantiate a `meta.event` object directly. Obtain a `meta.event` object from the `meta.class` `EventList` property, which contains an array of `meta.event` objects, one for each event defined by the class. For example, replace *`ClassName`* with the name of the class whose events you want to query:

```
mco = ?ClassName;
elist = mco.EventList;
elist(1).Name; % name of first event in list
```

Use the `metaclass` function to obtain a `meta.class` object from a class instance:

```
mco = metaclass(obj);
```

Properties

Property	Purpose
`Name` read only	Name of the event.
`Description` read only	Currently not used
`DetailedDescription` read only	Currently not used

<table>
<tr><th>Property</th><th>Purpose</th></tr>
<tr><td><code>Hidden</code></td><td>If <code>true</code>, the event does not appear in the list of events returned by the <code>events</code> function (or other event listing functions or viewers)</td></tr>
<tr><td><code>ListenAccess</code></td><td>Determines where you can create listeners for the event.

• <code>public</code> — unrestricted access
• <code>protected</code> — access from methods in class or subclasses
• <code>private</code> — access by class methods only (not from subclasses)
• List classes that have listen access to this event. Specify classes as <code>meta.class</code> objects in the form:
  - A single <code>meta.class</code> object
  - A cell array of <code>meta.class</code> objects. An empty cell array, <code>{}</code>, is the same as <code>private</code> access.

See “Controlling Access to Class Members”</td></tr>
</table>

Property	Purpose
NotifyAccess	Determines where code can trigger the event. • public — any code can trigger event • protected — can trigger event from methods in class or subclasses • private — can trigger event by class methods only (not from subclasses) • List classes that have notify access to this event. Specify classes as meta.class objects in the form: - A single meta.class object - A cell array of meta.class objects. An empty cell array, {}, is the same as private access. See "Controlling Access to Class Members"
DefiningClass	The meta.class object representing the class that defines this event.

See Also

meta.class | meta.property | meta.method | metaclass

How To

- "Events"
- "Getting Information About Classes and Objects"

Superclasses Heterogeneous

Purpose Superclass for MATLAB object metadata

Description The `meta.MetaData` class of objects represent MATLAB class definitions and the constituent parts of those definitions, such as properties and methods. Metadata enable a program to get information about a class definition.

The `meta.MetaData` class forms the root of the metadata class hierarchy, which enables the formation of arrays of metadata objects belonging to different specific classes.

MATLAB uses instances of the `meta.MetaData` class as the default object to fill in missing array elements.

`findobj` and `findprop`, can search the metadata hierarchy and return an array of different metadata objects. These function require the ability to form heterogeneous arrays containing various metaclass objects.

See the `matlab.mixin.Heterogeneous` class for more information on heterogeneous hierarchies.

Construction You cannot create an instance of the `meta.MetaData` class directly. MATLAB constructs instances of this class as required.

Copy Semantics Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples This example shows how the `meta.MetaData` class facilitates working with metaclasses.

Create a `meta.class` instance representing the MATLAB `timeseries` class:

```
>> mc = ?timeseries;
```

MATLAB uses `meta.MetaData` objects to fill empty array elements:

```
>> m(2) = mc
>> class(m(1))

ans =

meta.MetaData
>> class(m(2))

ans =

meta.class
```

Use `findobj` to find all properties and methods that have protected access:

```
>> protectedMembers = findobj(mc,{'Access','protected'},...
'-or',{'SetAccess','protected'},...
'-or',{'GetAccess','protected'});
```

The `timeseries` class defines both properties and methods that have protected access. Therefore, `findobj` returns a heterogeneous array of class `meta.MetaData`. This array contains both `meta.property` and `meta.method` objects.

```
>> protectedMembers

protectedMembers =

  11x1 heterogeneous meta.MetaData (meta.property, meta.method)
  handle with no properties.
  Package: meta
>> class(protectedMembers(1))

ans =

meta.property

>> protectedMembers(1).Name
```

```
ans =

Length
>> protectedMembers(1).SetAccess

ans =

protected
>> protectedMembers(1).GetAccess

ans =

public
```

See Also

handle | matlab.mixin.Heterogeneous

How To

- Class Attributes
- Property Attributes
- "Getting Information About Classes and Objects"

meta.method

Purpose

`meta.method` class describes MATLAB class methods

Description

The `meta.method` class provides information about the methods of MATLAB classes. The read/write properties of the `meta.method` class correspond to method attributes and are specified only from within class definitions.

You can query the read-only properties of the `meta.method` object to obtain information that is specified syntactically by the class (for example, to obtain the name of the class defining a method).

You cannot instantiate a `meta.method` object directly. Obtain a `meta.method` object from the `meta.class` `MethodList` property, which contains an array of `meta.method` objects, one for each class method. For example, replace *`ClassName`* with the name of the class whose methods you want to query:

```
mco = ?ClassName;
mlist = mco.MethodList;
mlist(1).Name; % name of first method in the list
```

Use the `metaclass` function to obtain a `meta.class` object from a class instance:

```
mco = metaclass(obj);
```

Properties

Property	Purpose
`Abstract`	If `true`, the method has no implementation. The method has a syntax line that can include arguments, which subclasses use when implementing the method. • Subclasses are not required to define the same number of input and output arguments.

Property	Purpose
	• The method can have comments after the `function` line • Does not contain `function` or `end` keywords, only the function syntax (e.g., `[a,b] = myMethod(x,y)`)
`Access` attribute, default = `public`	Determines what code can call this method. • `public` — unrestricted access • `protected` — access from methods in class
`DefiningClass`	The `meta.class` object representing the class that defines this method.
`Description` read only	Currently not used
`DetailedDescription` read only	Currently not used
`Hidden` attribute, default = `false`	When `false`, the method name shows in the list of methods displayed using the `methods` or `methodsview` commands. If set to `true`, the method name is not included in these listings.
`Name` read only	Name of the method.

Property	Purpose
`Sealed` attribute, default = `false`	If `true`, the method cannot be redefined in a subclass. Attempting to define a method with the same name in a subclass causes an error.

`Static` attribute, default = `false`	Set to `true` to define a method that does not depend on an object of the class and does not require an object argument. Call static methods using the class name in place of the object:
	
	`classname.methodname()`
	
	Or with an instance of the class, like any method:
	
	`o.methodname()`
	
	See "Static Methods"

See Also

`meta.class` | `meta.property` | `meta.event` | `metaclass`

How To

- "Methods"
- "Getting Information About Classes and Objects"

Purpose

meta.package class describes MATLAB packages

Description

The meta.package class contains information about MATLAB packages.

You cannot instantiate a meta.package object directly. Obtain a meta.package object from the meta.class ContainingPackage property, which contains a meta.package object, or an empty object, if the class is not in a package.

Properties

Property	Purpose
Name read only	Name of the package associated with this meta.package object
ClassList read only	List of classes that are scoped to this package. An object array of meta.class objects.
Classes read only Use ClassList instead	List of classes that are scoped to this package. A cell array of meta.class objects.
FunctionList read only	List of functions that are scoped to this package. An object array of function handles.
Functions read only Use FunctionList instead	List of functions that are scoped to this package. A cell array of function handles.
PackageList read only	List of packages that are scoped to this package. An object array of meta.package objects.
Packages read only Use PackageList instead	List of packages that are scoped to this package. A cell array of meta.package objects.
ContainingPackage read only	A meta.package object describing the package within which this package is contained, or an empty object if this package is not nested.

Methods

Method	Purpose
fromName	Static method returns a meta.package object for a specified package name.
getAllPackages	Static method returns a cell array of meta.package objects representing all top-level packages.

See Also

meta.class | meta.property | meta.method | meta.event

How To

- "Getting Information About Classes and Objects"

Purpose

Find abstract methods and properties

Syntax

```
meta.abstractDetails(ClassName)
meta.abstractDetails(mc)
absMembers = meta.abstractDetails( ___ )
```

Description

`meta.abstractDetails(ClassName)` displays a list of abstract methods and properties for the class with name `ClassName`. Use the fully specified name for classes in packages. MATLAB displays all public and protected abstract methods and properties, including those declared `Hidden`.

`meta.abstractDetails(mc)` displays a list of abstract methods and properties for the class represented by the `meta.class` object `mc`.

`absMembers = meta.abstractDetails( ___ )` returns an array of the metaclass objects corresponding to the abstract members of the class, and can include any of the input arguments in previous syntaxes. If the class has both abstract methods and abstract properties, `absMembers` is a heterogeneous array of class `meta.MetaData` containing `meta.method` and `meta.property` objects.

A class can be abstract without defining any abstract methods or properties if it declares the `Abstract` class attribute. In this case, `meta.abstractDetails` returns no abstract members for that class, but the class is abstract. See "Determine If a Class Is Abstract" for more information.

Input Arguments

ClassName

Name of the class specified as a character string (for example, `'MyClass'`)

mc

`meta.class` object representing the class (for example, `?MyClass`)

Output Arguments

absMembers

Array of metaclass objects representing abstract class members

Examples

Display Abstract Member Names

Define the class, `AbsBase`, with an abstract property:

```
classdef AbsBase
   properties (Abstract)
      Prop1
   end
   methods(Abstract)
      result = methodOne(obj)
      output = methodTwo(obj)
   end
end
```

Pass the class name (`AbsBase`) as a string:

```
meta.abstractDetails('AbsBase')
```

`meta.abstractDetails` displays the names of the abstract properties and methods defined in the class `AbsBase`.

```
Abstract methods for class AbsBase:
    methodTwo   % defined in AbsBase
    methodOne   % defined in AbsBase

Abstract properties for class AbsBase:
    Prop1   % defined in AbsBase
```

Return Abstract Member Metaclass Objects

Pass a `meta.class` object representing the `AbsBase` class and return the metaclass objects for the abstract members. Use the definition of the `AbsBase` class from the previous example.

```
mc = ?AbsBase;
```

```
absMembers = meta.abstractDetails(mc);
```

`absMembers` is a heterogeneous array containing a `meta.property` object for the `Prop1` abstract property and `meta.method` objects for the `methodOne` and `methodTwo` abstract methods.

List the names of the metaclass objects.

```
for k=1:length(absMembers)
   disp(absMembers(k).Name)
end

methodTwo
methodOne
Prop1
```

Find Inherited Abstract Members

Derive the `SubAbsBase` class from `AbsBase`, which is defined in a previous example.

```
classdef SubAbsBase < AbsBase
   properties
      SubProp = 1;
   end
   methods
      function result = methodOne(obj)
         result = obj.SubProp + 1;
      end
   end
end
```

Display the names of the abstract members inherited by `SubAbsBase`.

```
meta.abstractDetails('SubAbsBase')

Abstract methods for class SubAbsBase:
    methodTwo   % defined in AbsBase
```

```
Abstract properties for class SubAbsBase:
    Prop1   % defined in AbsBase
```

To make `SubAbsBase` a concrete class, you need to implement concrete versions of `methodTwo` and `Prop1` in the subclass.

See Also

`meta.class | meta.class.fromName`

Concepts

- "Defining Abstract Classes"
- "Getting Information About Classes and Objects"

meta.package.fromName

Purpose

Return `meta.package` object for specified package

Syntax

```
mpkg = meta.package.fromName('pkgname')
```

Description

`mpkg = meta.package.fromName('pkgname')` is a static method that returns the `meta.package` object associated with the named package. If `pkgname` is a nested package, then you must provide the fully qualified name (e.g., `'pkgname1.pkgname2'`).

Examples

List the classes in the `event` package:

```
mev = meta.package.fromName('event');
for k=1:length(mev.Classes)
   disp(mev.Classes{k}.Name)
end
event.EventData
event.PropertyEvent
event.listener
event.proplistener
```

See Also

`meta.package` | `meta.package.getAllPackages`

meta.package.getAllPackages

Purpose

Get all top-level packages

Syntax

```
P = meta.package.getAllPackages
```

Description

`P = meta.package.getAllPackages` is a static method that returns a cell array of `meta.package` objects representing all the top-level packages that are visible on the MATLAB path or defined as top-level built-in packages. You can access subpackages using the `Packages` property of each `meta.package` object.

Note that the time required to find all the packages on the path might be excessively long in some cases. You should therefore avoid using this method in any code where execution time is a consideration. `getAllPackages` is generally intended for interactive use only.

See Also

`meta.package` | `meta.package.fromName`

Purpose

`meta.property` class describes MATLAB class properties

Description

The `meta.property` class provides information about the properties of MATLAB classes. The read/write properties of the `meta.property` class correspond to property attributes and are specified only from within your class definitions.

You can query the read-only properties of the `meta.property` object to obtain information that is specified syntactically by the class (for example, to obtain the function handle of a property's set access method).

You cannot instantiate a `meta.property` object directly. Obtain a `meta.property` object from the `meta.class` `PropertList` property, which contains an array of `meta.property` objects, one for each class property. For example, replace *`ClassName`* with the name of the class whose properties you want to query:

```
mco = ?ClassName;
plist = mco.PropertyList;
plist(1).Name; % name of first property
```

Use the `metaclass` function to obtain a `meta.class` object from a class instance:

```
mco = metaclass(obj);
```

Properties

Property	Purpose
`Name` read only	Name of the property.
`Description` read only	Currently not used
`DetailedDescription` read only	Currently not used

Property	Purpose
AbortSet attribute, default = false	If true, and this property belongs to a handle class, then MATLAB does not set the property value if the new value is the same as the current value. This prevents the triggering of property PreSet and PostSet events. See "Listen for Changes to Property Values"
Abstract attribute, default = false	If true, the property has no implementation, but a concrete subclass must redefine this property without Abstract being set to true. • Abstract properties cannot define set or get access methods. See "Property Access Methods" • Abstract properties cannot define initial values. "Assigning a Default Value" • All subclasses must specify the same values as the superclass for the property SetAccess and GetAccess attributes. • Abstract=true should be used with the class attribute Sealed=false (the default).
Access	public – unrestricted access protected – access from class or subclasses private – access by class members only (not subclasses) List of classes that have get and set access to this property. Specify classes as meta.class objects in the form: • A single meta.class object

Property	Purpose
	• A cell array of `meta.class` objects. An empty cell array, `{}`, is the same as `private` access. Use `Access` to set both `SetAccess` and `GetAccess` to the same value. Query the values of `SetAccess` and `GetAccess` directly (not `Access`).
`Constant` attribute, default = `false`	Set to `true` if you want only one value for this property in all instances of the class. • Subclasses inherit constant properties, but cannot change them. • `Constant` properties cannot be `Dependent` • `SetAccess` is ignored. See "Properties with Constant Values"
`DefaultValue`	Property default value (if specified in class definition). See also `HasDefault` property. Abstract, dependent and dynamic properties cannot specify default values.
`DefiningClass`	The `meta.class` object representing the class that defines this property.

<table>
<tr><th>Property</th><th>Purpose</th></tr>
<tr><td><code>GetAccess</code> attribute, default = <code>public</code></td><td> <code>public</code> – unrestricted access <code>protected</code> – access from class or subclasses <code>private</code> – access by class members only List classes that have get access to this property. Specify classes as <code>meta.class</code> objects in the form: A single <code>meta.class</code> object
A cell array of <code>meta.class</code> objects. An empty cell array, <code>{}</code>, is the same as <code>private</code> access. See “Controlling Access to Class Members” </td></tr>
<tr><td><code>Dependent</code> attribute, default = <code>false</code></td><td> If <code>false</code>, property value is stored in object. If <code>true</code>, property value is not stored in object and the set and get functions cannot access the property by indexing into the object using the property name. See “Property Get Methods” </td></tr>
<tr><td><code>Transient</code> attribute, default = <code>false</code></td><td>If <code>true</code>, property value is not saved when object is saved to a file. See “Understanding the Save and Load Process” for more about saving objects.</td></tr>
<tr><td><code>GetMethod</code> read only</td><td>Function handle of the get method associated with this property. Empty if there is no get method specified. See “Get Method Syntax”</td></tr>
</table>

Property	Purpose
`GetObservable` attribute, default = `false`	If `true`, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are queried. See "Property-Set and Query Events"
`HasDefault`	Property contains a boolean value indicating if the property defines a default value. Test `HasDefault` before querying the `DefaultValue` property to avoid a `MATLAB:class:NoDefaultDefined` error.
`Hidden` attribute, default = `false`	Determines whether the property should be shown in a property list (e.g., Property Inspector, call to `properties`, etc.).
`SetAccess` attribute, default = `public`	`public` – unrestricted access `protected` – access from class or subclasses `private` – access by class members only `immutable` — property can be set only in the constructor. See "Mutable and Immutable Properties" List classes that have set access to this property. Specify classes as `meta.class` objects in the form: • A single `meta.class` object • A cell array of `meta.class` objects. An empty cell array, `{}`, is the same as `private` access. See "Controlling Access to Class Members"

Property	Purpose
`SetMethod` read only	Function handle of the set method associated with this property. Empty if there is no set method specified. See "Property Set Methods"
`SetObservable` attribute, default = `false`	If `true`, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are modified. See "Property-Set and Query Events"

Events

See "Listen for Changes to Property Values" for information on using property events.

Event Name	Purpose
`PreGet`	Event occurs just before property is queried.
`PostGet`	Event occurs just after property has been queried
`PreSet`	Event occurs just before this property is modified
`PostSet`	Event occurs just after this property has been modified

See Also

`meta.class` | `meta.method` | `meta.event` | `metaclass`

How To

- "Properties"
- "Getting Information About Classes and Objects"

Purpose

Obtain `meta.class` object

Syntax

```
mc = metaclass(object)
mc = ?ClassName
```

Description

`mc = metaclass(object)` returns the `meta.class` object for the class of `object`. The `object` input argument can be a scalar or an array of objects. However, `metaclass` always returns a scalar `meta.class` object.

`mc = ?`*`ClassName`* returns the `meta.class` object for the class with name, *`ClassName`*. The `?` operator works only with a class name, not an object.

If you pass a class name as a string to the `metaclass` function, it returns the `meta.class` object for the `char` class. Use the `?` operator or the `meta.class.fromName` method to obtain the `meta.class` object from a class name. Use this method if you want to pass the class name in a string variable.

Examples

Return the `meta.class` object for an instance of the `MException` class:

```
obj = MException('Msg:ID','MsgTxt');
mc = metaclass(obj);
```

Use the `?` operator to get the `meta.class` object for the `hgsetget` class:

```
mc = ?hgsetget;
```

See Also

`meta.class` | `meta.class.fromName`

Tutorials

- “Class Metadata”

methods

Purpose

Class method names

Syntax

```
methods('classname')
methods(...,'-full')
m = methods(...)
```

Description

`methods('`*`classname`*`')` displays the names of the methods for the class *`classname`*. If *`classname`* is a MATLAB or Java class, then `methods` displays only public methods, including those methods inherited from superclasses.

`methods(...,'-full')` displays a full description of the methods, including inheritance information and, for MATLAB and Java methods, method attributes and signatures. `methods` does not remove duplicate method names with different signatures. Do not use this option with classes defined before MATLAB 7.6.

`m = methods(...)` returns the method names in a cell array of strings.

`methods` is also a MATLAB class-definition keyword. See `classdef` for more information on class-definition keywords.

This function does not show generic methods from classes based on the Microsoft .NET Framework. Use your product documentation to get information on generic methods.

Examples

Retrieve the names of the static methods in class `MException`:

```
methods('MException')

Methods for class MException:

addCause       getReport      ne             throw
eq             isequal        rethrow        throwAsCaller

Static methods:

last
```

See Also

`methodsview` | `properties` | `events` | `what` | `which`

Tutorials

- "Methods"

methodsview

Purpose

View class methods

Syntax

```
methodsview packagename.classname
methodsview classname
methodsview(object)
```

Description

`methodsview packagename.classname` displays information about the methods in the class, `classname`. If the class is in a package, include `packagename`. If `classname` is a MATLAB or Java class, `methodsview` lists only public methods, including those methods inherited from superclasses.

`methodsview classname` displays information describing the class `classname`.

`methodsview(object)` displays information about the methods of the class of `object`.

`methodsview` creates a window that displays the methods defined in the specified class. `methodsview` provides additional information like arguments, returned values, and superclasses. It also includes method qualifiers (for example, `abstract` or `synchronized`) and possible exceptions thrown.

Examples

List information on all methods in the `java.awt.MenuItem` class:

```
methodsview java.awt.MenuItem
```

MATLAB displays this information in a new window.

See Also

`methods` | `import` | `class` | `javaArray`

Purpose

Compile MEX-function from C/C++ or Fortran source code

Syntax

```
mex -help
mex -setup
mex filenames
mex options filenames
```

Description

`mex -help` displays the help file for `mex`.

`mex -setup` lets you select or change the compiler configuration. MATLAB software searches for installed compilers and allows you to choose an options file as the default for future invocations of `mex`. For more information, see "Build MEX-Files" and `mex.getCompilerConfigurations`.

`mex` *`filenames`* compiles and links one or more C/C++ or Fortran source files specified in `filenames` into a shared library called a binary MEX-file from MATLAB.

`mex` *`options filenames`* compiles and links one or more source files specified in `filenames` using one or more of the specified command-line `options`.

The MEX-file has a platform-dependent extension. Use the `mexext` function to return the extension for the current machine or for all supported platforms.

`filenames` can be any combination of source files, Simulink S-function files, object files, and library files. Include both the file name and the file extension in `filenames`. A non-source-code `filenames` parameter is passed to the linker without being compiled.

All valid command-line `options` are shown in the on page 1-3322 table. These options are available on all platforms except where noted.

`mex` also can build executable files for stand alone MATLAB engine and MAT-file applications. For more information, see "Engine/MAT Stand Alone Application Details" on page 1-3327.

You can run `mex` from the MATLAB Command Prompt, the Microsoft Windows Command Prompt, or the UNIX shell. `mex` is a script named

`mex.bat` on Windows systems and `mex` on UNIX systems. It is located in the *matlabroot*`/bin` folder.

The first file listed in `filenames` becomes the name of the binary MEX-file. You can list other source, object, or library files as additional `filenames` parameters to satisfy external references.

`mex` uses an options file to specify variables and values that are passed as arguments to the compiler, linker, and other tools (for example, the resource linker on Windows systems). For more information, see "Options File Details" on page 1-3326. The default name for the options file is `mexopts.bat` (Windows systems) or `mexopts.sh` (UNIX systems).

Command-line options to `mex` may supplement or override contents of the options file. For more information, see "Override Option Details" on page 1-3327.

For an up-to-date list of supported compilers, see the Supported and Compatible Compilers Web page.

MEX Script Switches

Switch	Function
`@`*`rsp_file`*	(Windows systems only) Include the contents of the text file *`rsp_file`* as command-line arguments to `mex`.
`-`*`arch`*	Build an output file for architecture *`arch`*. To determine the value for *`arch`*, type `computer('arch')` at the MATLAB Command Prompt on the target machine. Valid values for *`arch`* depend on the architecture of the build platform.
`-c`	Compile only. Creates an object file, but not a binary MEX-file.

(Continued)

Switch	Function
`-compatibleArrayDims`	Build a binary MEX-file using the MATLAB Version 7.2 array-handling API, which limits arrays to 2^{31}-1 elements. Default option. (See the `-largeArrayDims` option.) In verbose mode, if you do not specify either the `-largeArrayDims` or the `-compatibleArrayDims` switches, MATLAB displays a message showing the default switch.
`-cxx`	(UNIX systems only) Use the C++ linker to link the MEX-file if the first source file is in C and there are one or more C++ source or object files. This option overrides the assumption that the first source file in the list determines which linker to use.
`-D`*`name`*	Define a symbol name to the C preprocessor. Equivalent to a `#define` *`name`* directive in the source. Do not add a space after this switch.
`-D`*`name=value`*	Define a symbol name and value to the C preprocessor. Equivalent to a `#define` *`name value`* directive in the source. Do not add a space after this switch.
`-f` *`optionsfile`*	Specify location and name of options file to use. Overrides the `mex` default-options-file search mechanism.

(Continued)

Switch	Function
`-fortran`	(UNIX systems only) Specify that the gateway routine is in Fortran. This option overrides the assumption that the first source file in the list determines which linker to use.
`-g`	Create a binary MEX-file containing additional symbolic information for use in debugging. This option disables the `mex` default behavior of optimizing built object code (see the `-O` option).
`-h[elp]`	Print help for `mex`.
`-I`*`pathname`*	Add *`pathname`* to the list of folders to search for `#include` files. Do not add a space after this switch.
`-inline`	This option has been removed.
`-l`*`name`*	Link with object library. On Windows systems, *`name`* expands to *`name`*`.lib` or `lib`*`name`*`.lib` and on UNIX systems, to `lib`*`name`*`.so` or `lib`*`name`*`.dylib`. Do not add a space after this switch.
`-L`*`folder`*	Add *`folder`* to the list of folders to search for libraries specified with the `-l` option. On UNIX systems, you must also set the run-time library path, as explained in "Setting Run-Time Library Path". Do not add a space after this switch.

(Continued)

Switch	Function
`-largeArrayDims`	Build a binary MEX-file using the MATLAB large-array-handling API. This API can handle arrays with more than 2^{31}-1 elements. (See the `-compatibleArrayDims` option.) In verbose mode, if you do not specify either the `-largeArrayDims` or the `-compatibleArrayDims` switches, MATLAB displays a message showing the default switch.
`-n`	No execute mode. Print any commands that `mex` would otherwise have executed, but do not actually execute any of them.
`-O`	Optimize the object code. Optimization is enabled by default and by including this option on the command line. If the `-g` option appears without the `-O` option, optimization is disabled.
`-outdir` *`dirname`*	Place all output files in folder *`dirname`*.
`-output` *`resultname`*	Create binary MEX-file named *`resultname`*. Automatically appends the appropriate MEX-file extension. Overrides the default MEX-file naming mechanism.
`-setup`	Specify the compiler options file to use when calling the `mex` function. When you use this option, all other command-line options are ignored.

(Continued)

Switch	Function
`-U`*`name`*	Remove any initial definition of the C preprocessor symbol *`name`*. (Inverse of the `-D` option.) Do not add a space after this switch.
`-v`	Verbose mode. Print the values for important internal variables after the options file is processed and all command-line arguments are considered. Prints each compile step and final link step fully evaluated.
`name``=`*`value`*	Override an options file variable for variable *`name`*. For examples, see Override Option Details on `mex` reference page.

Tips

Matrix Library API

For a list of the mxArray functions in the C/C++ and Fortran API Reference, see:

- "C/C++ Matrix Library"
- "Fortran Matrix Library"

Options File Details

MATLAB provides template options files for the compilers that are supported by `mex`. These templates are located in the *`matlabroot`*`\bin\win32\mexopts` or the *`matlabroot`*`\bin\win64\mexopts` folders on Windows systems, or the *`matlabroot`*`/bin` folder on UNIX systems. These template options files are used by the `-setup` option to define the selected default options file.

Override Option Details

Use the *name=value* command-line argument to override a variable specified in the options file at the command line. When using this option, you may need to use the shell's quoting syntax to protect characters such as spaces, which have a meaning in the shell syntax.

This option is processed after the options file is processed and all command line arguments are considered.

On Windows platforms, at either the MATLAB prompt or the DOS prompt, use double quotes ("). For example:

```
mex -v COMPFLAGS="$COMPFLAGS -Wall" LINKFLAGS="$LINKFLAGS /VERBOSE" yp
```

At the MATLAB command line on UNIX platforms, use double quotes ("). Use the backslash (\) escape character before the dollar sign ($). For example:

```
mex -v CFLAGS="\$CFLAGS -Wall" LDFLAGS="\$LDFLAGS -w" yprime.c
```

At the shell command line on UNIX platforms, use single quotes ('). For example:

```
mex -v CFLAGS='$CFLAGS -Wall' LDFLAGS='$LDFLAGS -w' yprime.c
```

Engine/MAT Stand Alone Application Details

`mex` can build executable files for stand alone MATLAB engine and MAT-file applications. For these applications, `mex` does not use the default options file; you must use the `-f` option to specify an options file.

The options files used to generate stand alone MATLAB engine and MAT-file executables are named `*engmatopts.bat` on Windows systems, or `engopts.sh` and `matopts.sh` on UNIX systems, and are located in the same folder as the template options files referred to above in Options File Details.

Examples

Compiling a C File

Copy the yprime.c example to a writable folder on your path, such as c:\work.

```
copyfile(fullfile(matlabroot,'extern','examples','mex',...
  'yprime.c'), fullfile('c:','work'));
```

Compile yprime.c.

```
mex yprime.c
```

Using Verbose Mode

When debugging, it is often useful to use verbose mode, as well as include symbolic debugging information:

```
mex -v -g yprime.c
```

Overriding Command Line Options

For examples, see "Override Option Details" on page 1-3327.

List of Examples

For a list of MEX example files available with MATLAB, see "Table of MEX-File Source Code Files".

See Also

computer | dbmex | inmem | loadlibrary | mexext | pcode | prefdir | system | mex.getCompilerConfigurations | clear

Purpose

Get compiler configuration information for building MEX-files

Syntax

```
cc = mex.getCompilerConfigurations()
cc = mex.getCompilerConfigurations('lang')
cc = mex.getCompilerConfigurations('lang','list')
```

Description

`cc = mex.getCompilerConfigurations()` returns a `mex.CompilerConfiguration` object `cc` containing information about the selected compiler configuration used by `mex`. The selected compiler is the one you choose when you run the `mex -setup` command. For details about the `mex.CompilerConfiguration` class, see "mex.CompilerConfiguration" on page 1-3330.

`cc = mex.getCompilerConfigurations('`*`lang`*`')` returns an array of `mex.CompilerConfiguration` objects `cc` containing information about the selected configuration for the given *`lang`*. If the language of the selected complier is different from *`lang`*, then `cc` is empty.

Language *`lang`* is a string with one of the following values:

- **`'Any'`** — All supported languages. This is the default value.
- **`'C'`** — All C compiler configurations, including C++ configurations.
- **`'C++'`** or **`'CPP'`** — All C++ compiler configurations.
- **`'Fortran'`** — All Fortran compiler configurations.

`cc = mex.getCompilerConfigurations('`*`lang`*`','`*`list`*`')` returns an array of `mex.CompilerConfiguration` objects `cc` containing information about configurations for the given language and the given *`list`*. Values for *`list`* are:

- **`'Selected'`** — The compiler you chose when you ran `mex-setup`. This is the default value.
- **`'Installed'`** — All supported compilers `mex` finds installed on your system.
- **`'Supported'`** — All compilers supported in the current release. For an up-to-date list of supported compilers, see the Supported and Compatible Compilers Web page.

Classes

mex.CompilerConfiguration

The `mex.CompilerConfiguration` class contains the following read-only properties about compiler configurations.

Property	Purpose
`Name`	Name of the compiler
`Manufacturer`	Name of the manufacturer of the compiler
`Language`	Compiler language
`Version`	(Windows platforms only) Version of the compiler
`Location`	(Windows platforms only) Folder where compiler is installed
`Details`	Additional read-only properties about the compiler configuration. These properties can differ across compilers, platforms, and releases of MATLAB.

Examples

Selected Compiler Example

```
myCompiler = mex.getCompilerConfigurations()
```

MATLAB software displays information similar to the following (depending on your architecture, your version of MATLAB, and what you selected when you ran `mex -setup`):

```
myCompiler =

  mex.CompilerConfiguration
  Package: mex

  Properties:
            Name: 'Microsoft Visual C++ 2005'
    Manufacturer: 'Microsoft'
        Language: 'C++'
         Version: '8.0'
```

```
        Location: '%VS80COMNTOOLS%\..\..'
         Details: [1x1 mex.CompilerConfigurationDetails]

  Methods
```

Supported Compiler Configurations Example

```
allCC = mex.getCompilerConfigurations('Any','Supported')
```

MATLAB displays information similar to the following:

```
allCC =

  1x11 mex.CompilerConfiguration
  Package: mex

  Properties:
    Name
    Manufacturer
    Language
    Version
    Location
    Details

  Methods
```

This version of MATLAB supports eleven configurations, hence, `allCC` is a 1-by-11 matrix.

Supported C Compilers Example

To see what C compilers MATLAB supports, type:

```
cLanguageCC = mex.getCompilerConfigurations('C','Supported')
```

MATLAB displays the following information (the number of compilers for your version of MATLAB may be different):

```
cLanguageCC =

  1x9 mex.CompilerConfiguration
  Package: mex

  Properties:
    Name
    Manufacturer
    Language
    Version
    Location
    Details

  Methods
```

To display the compiler names, type:

```
format compact
cLanguageCC.Name
```

MATLAB displays information similar to the following:

```
ans =
Intel C++
ans =
Lcc-win32
ans =
Microsoft Visual C++
ans =
Microsoft Visual C++ 2003
ans =
Microsoft Visual C++ 2005 Express Edition
ans =
Microsoft Visual C++ 2005
ans =
Microsoft Visual C++ 2008
ans =
```

```
Open WATCOM C/C++
```

Example — Viewing Build Options for a C Compiler

To see what build options MATLAB uses with a particular C compiler, create an array CC of all supported C compiler configurations:

```
CC = mex.getCompilerConfigurations('C','Supported');
disp('Compiler Name')
for i = 1:3; disp(CC(i).Name); end;
```

MATLAB displays a list similar to:

```
Intel C++
Lcc-win32
Microsoft Visual C++
```

To see the build options for the Microsoft Visual C++ compiler, type:

```
CC(3).Details
```

MATLAB displays information similar to the following (output is formatted):

```
ans =
  mex.CompilerConfigurationDetails
  Package: mex

  Properties:
         CompilerExecutable: 'cl'
              CompilerFlags: '-c -Zp8 -G5 -W3 -EHs
                             -DMATLAB_MEX_FILE -nologo /MD'
          OptimizationFlags: '-O2 -Oy- -DNDEBUG'
                 DebugFlags: '-Zi
                             -Fd"%OUTDIR%%MEX_NAME%.pdb"'
           LinkerExecutable: 'link'
                LinkerFlags: [1x258 char]
    LinkerOptimizationFlags: ''
```

```
            LinkerDebugFlags: '/debug'
    Methods
```

See Also

`mex`

Purpose

Capture error information

Syntax

```
exception = MException(msgIdent, msgString, v1, v2, ..., vN)
```

Description

`exception = MException(msgIdent, msgString, v1, v2, ..., vN)` captures information about a specific error and stores it in MException object, *exception*. Information stored in the object includes a message identifier *msgIdent* and an error message string *msgString*. Optional arguments v_1, v_2, ... represent text or numeric values that replace conversion specifiers in *msgString* at run time.

Message identifier *msgIdent* is a character string composed of at least two substrings, the *component* and the *mnemonic*, separated by a colon (e.g., *component*:*mnemonic*). The purpose of the identifier is to better identify the source of the error. See the documentation on "Message Identifiers" for more information.

Message string *msgString* is a character string that informs the user about the cause of the error and can also suggest how to correct the faulty condition. The *msgString* string can include escape sequences such as `\t` or `\n`, as well as any of the format specifiers supported by the `sprintf` function (such as `%s` or `%d`). Additional arguments v_1, v_2, ..., v_N provide values that correspond to and replace the conversion specifiers.

For example, if *msgString* is "Error on line `%d`, command `%s`", then v_1 is the line number at which the error was detected, and v_2 is the command that failed. See "Formatting Strings" in the MATLAB Programming Fundamentals documentation for more detailed information on using string formatting commands.

The *exception* output is an object of the `MException` class. `MException` is the constructor for this class. In addition to calling the constructor directly, you can also create an object of `MException` with any of the following functions: `error` and `assert`. See the documentation and figure in the section "The MException Class" for more information on this class.

Properties

The MException object has four properties: identifier, message, stack, and cause. Click any of the links below to find out more about MException properties:

Property	Description
identifier	Identifies the error.
message	Formatted error message that is displayed.
stack	Structure containing stack trace information such as file name, function name, and line number where the MException was thrown.
cause	Cell array of MException that caused this exception to be created.

Methods

The MException class has the following methods. Click any of the links below to find out more about MException methods:

Method	Description
addCause	Appends an MException to the cause field of another MException.
eq	Tests scalar MException objects for equality.
getReport	Returns a formatted message string that uses the same format as errors thrown by internal MATLAB code.
isequal	Tests scalar MException objects for equality.
last	Returns an MException object for the most recently thrown exception.
ne	Tests scalar MException objects for inequality.

Method	Description
`rethrow`	Reissues an exception that has been caught, causing the program to stop.
`throw`	Issues an exception from the currently running function.
`throwAsCaller`	Issues an exception from the currently running function, also omitting the current stack frame from the `stack` field of the `MException`.

Tips

Valid escape sequences for the `msgString` argument are `\b`, `\f`, `\n`, `\r`, `\t`, and `\x` or `\` when followed by a valid hexadecimal or octal number, respectively. Following a backslash in the `msgString` with any other character causes MATLAB to issue a warning. Conversion specifiers are similar to those used in the C programming language and in the `sprintf` function.

All string input arguments must be enclosed in single quotation marks.

Examples

Example 1 — Formatted Messages

If your message string requires formatting specifications like those used with the `sprintf` function, you can use this syntax to compose the error message string:

```
exception = MException(msgIdent, msgString, v1, v2, ...)
```

For example,

```
exception = MException('AcctError:Incomplete', ...
      'Field ''%s.%s'' is not defined.', ...
      'Accounts', 'ClientName');

exception.message
ans =
    Field 'Accounts.ClientName' is not defined.
```

Example 2

The `catch` block in this example checks to see if the specified file could not be found. If this is the case, the program allows for the possibility that a common variation of the filename extension (e.g., `jpeg` instead of `jpg`) was used by retrying the operation with a modified extension. This is done using a `try/catch` statement that is nested within the original `try/catch`.

```
function d_in = read_image(filename)
[path name ext] = fileparts(filename);
try
   fid = fopen(filename, 'r');
   d_in = fread(fid);
catch exception

   % Did the read fail because the file could not be found?
   if ~exist(filename, 'file')

      % Yes. Try modifying the filename extension.
      switch ext
      case '.jpg'    % Change jpg to jpeg
          altFilename = strrep(filename, '.jpg', '.jpeg')
      case '.jpeg'   % Change jpeg to jpg
          altFilename = strrep(filename, '.jpeg', '.jpg')
      case '.tif'    % Change tif to tiff
          altFilename = strrep(filename, '.tif', '.tiff')
      case '.tiff'   % Change tiff to tif
          altFilename = strrep(filename, '.tiff', '.tif')
      otherwise
         rethrow(exception);
      end

      % Try again, with modifed filename.
      try
         fid = fopen(altFilename, 'r');
         d_in = fread(fid);
      catch
```

```
        rethrow(exception)
      end
   end
end
```

Example 3 — Nested try/catch

This example attempts to open a file in a folder that is not on the MATLAB path. It uses a nested `try-catch` block to give the user the opportunity to extend the path. If the file still cannot be found, the program issues an exception with the first error appended to the second:

```
function data = read_it(filename);
try
   % Attempt to open and read from a file.
   fid = fopen(filename, 'r');
   data = fread(fid);
catch exception1
   % If the error was caused by an invalid file ID, try
   % reading from another location.
   if strcmp(exception1.identifier, 'MATLAB:FileIO:InvalidFid')
      msg = sprintf( ...
         '\nCannot open file %s. Try another location?  ', ...
         filename);
      reply = input(msg, 's')
      if reply(1) == 'y'
          newFolder = input('Enter folder name:  ', 's');
      else
          throw(exception1);
      end
      oldpath = addpath(newFolder);
      try
         fid = fopen(filename, 'r');
         data = fread(fid);
      catch exception2
         exception3 = addCause(exception2, exception1)
         path(oldpath);
         throw(exception3);
```

```
      end
      path(oldpath);
   end
end
fclose(fid);
```

If you run this function in a `try`-`catch` block at the command line, you can look at the `MException` object by assigning it to a variable (`e`) with the `catch` command.

See Also

`try` | `catch` | `error` | `assert` | `throw(MException)` | `rethrow(MException)` | `throwAsCaller(MException)` | `addCause(MException)` | `getReport(MException)` | `eq(MException)` | `isequal(MException)` | `ne(MException)` | `last(MException)` | `dbstack`

How To

- "The MException Class"

Purpose

Binary MEX-file name extension

Syntax

```
ext = mexext
extlist = mexext('all')
```

Description

`ext = mexext` returns the file name extension for the current platform.

`extlist = mexext('all')` returns a `struct` with fields `arch` and `ext` describing MEX-file name extensions for all platforms.

Tips

For a table of file extensions, see "Using Binary MEX-Files".

Examples

Find the MEX-file extension for the system you are currently working on:

```
ext = mexext

ext =
   mexw32
```

Find the MEX-file extension for an Apple Macintosh system:

```
extlist = mexext('all');

for k=1:length(extlist)
   if strcmp(extlist(k).arch, 'maci64')
   disp(sprintf('Arch: %s      Ext: %s', ...
           extlist(k).arch, extlist(k).ext))
end, end

Arch: maci64       Ext: mexmaci64
```

See Also

`mex | computer`

mfilename

Purpose

File name of currently running function

Syntax

```
mfilename
p = mfilename('fullpath')
c = mfilename('class')
```

Description

`mfilename` returns a string containing the file name of the most recently invoked function. When called from within the file, it returns the name of that file. This allows a function to determine its name, even if the file name has been changed.

`p = mfilename('fullpath')` returns the full path and name of the file in which the call occurs, not including the filename extension.

`c = mfilename('class')` in a method, returns the class of the method, not including the leading `@` sign. If called from a nonmethod, it yields the empty string.

Tips

If `mfilename` is called with any argument other than the above two, it behaves as if it were called with no argument.

When called from the command line, `mfilename` returns an empty string.

To get the names of the callers of a MATLAB function file, use `dbstack` with an output argument.

See Also

`dbstack` | `function` | `nargin` | `nargout` | `inputname`

Purpose

Download files from FTP server

Syntax

```
mget(ftpobj,contents)
mget(ftpobj,contents,target)
```

Description

`mget(ftpobj,contents)` retrieves the file or folder specified by `contents` from an FTP server into the MATLAB current folder.

`mget(ftpobj,contents,target)` retrieves the file or folder into the local folder specified by `target`, which includes an absolute or relative path.

Input Arguments

ftpobj

FTP object created by `ftp`.

contents

String enclosed in single quotation marks that specifies either a file name or a folder name. Can include a wildcard character (*).

target

String enclosed in single quotation marks that specifies the absolute or relative path of the local folder to contain the downloaded contents.

Examples

Connect to an FTP server and retrieve the file `README` into the current MATLAB folder:

```
mw = ftp('ftp.mathworks.com');
mget(mw, 'README');
close(mw);
```

See Also

`cd` | `ftp` | `mput`

min

Purpose

Smallest elements in array

Syntax

```
C = min(A)
C = min(A,B)
C = min(A,[],dim)
[C,I] = min(...)
```

Description

`C = min(A)` returns the smallest elements along different dimensions of an array.

If `A` is a vector, `min(A)` returns the smallest element in `A`.

If `A` is a matrix, `min(A)` treats the columns of `A` as vectors, returning a row vector containing the minimum element from each column.

If `A` is a multidimensional array, `min` operates along the first nonsingleton dimension.

`C = min(A,B)` returns an array the same size as `A` and `B` with the smallest elements taken from `A` or `B`. The dimensions of `A` and `B` must match, or they may be scalar.

`C = min(A,[],dim)` returns the smallest elements along the dimension of `A` specified by scalar `dim`. For example, `min(A,[],1)` produces the minimum values along the first dimension of `A`.

`[C,I] = min(...)` finds the indices of the minimum values of `A`, and returns them in output vector `I`. If there are several identical minimum values, the index of the first one found is returned.

Examples

Return the minimum of a 2-by-3 matrix from each column:

```
X = [2 8 4; 7 3 9];
min(X,[],1)
ans =

     2     3     4
```

Return the minimum from each row:

```
min(X,[],2)
ans =

     2
     3
```

Compare each element of `X` to a scalar:

```
min(X,5)
ans =

     2     5     4
     5     3     5
```

Tips

For complex input `A`, `min` returns the complex number with the smallest complex modulus (magnitude), computed with `min(abs(A))`. Then computes the smallest phase angle with `min(angle(x))`, if necessary.

The `min` function ignores `NaN`s.

See Also

`max` | `mean` | `median` | `sort`

MinimizeCommandWindow

Purpose

Minimize size of Automation server window

Syntax

MATLAB Client

```
h.MinimizeCommandWindow
MinimizeCommandWindow(h)
```

IDL Method Signature

```
HRESULT MinimizeCommandWindow(void)
```

Microsoft Visual Basic Client

```
MinimizeCommandWindow
```

Description

`h.MinimizeCommandWindow` minimizes the window for the server attached to handle `h`, and makes it inactive.

`MinimizeCommandWindow(h)` is an alternate syntax.

If the server window was already in a minimized state, `MinimizeCommandWindow` does nothing.

Examples

From a MATLAB client, modify the size of the command window in a MATLAB Automation server:

```
h = actxserver('matlab.application');
h.MinimizeCommandWindow;
% Now return the server window to its former state on
% the desktop and make it the currently active window.
h.MaximizeCommandWindow;
```

From a Visual Basic .NET client, modify the size of the command window in a MATLAB Automation server:

```
Dim Matlab As Object

Matlab = CreateObject("matlab.application")
Matlab.MinimizeCommandWindow
```

```
'Now return the server window to its former state on
'the desktop and make it the currently active window.

Matlab.MaximizeCommandWindow
```

See Also

`MaximizeCommandWindow`

How To

- “MATLAB COM Automation Server Interface”
- “Controlling the Server Window”

Purpose

Minimum residual method

Syntax

```
x = minres(A,b)
minres(A,b,tol)
minres(A,b,tol,maxit)
minres(A,b,tol,maxit,M)
minres(A,b,tol,maxit,M1,M2)
minres(A,b,tol,maxit,M1,M2,x0)
[x,flag] = minres(A,b,...)
[x,flag,relres] = minres(A,b,...)
[x,flag,relres,iter] = minres(A,b,...)
[x,flag,relres,iter,resvec] = minres(A,b,...)
[x,flag,relres,iter,resvec,resveccg] = minres(A,b,...)
```

Description

`x = minres(A,b)` attempts to find a minimum norm residual solution `x` to the system of linear equations `A*x=b`. The `n`-by-`n` coefficient matrix `A` must be symmetric but need not be positive definite. It should be large and sparse. The column vector `b` must have length `n`. You can specify `A` as a function handle, `afun`, such that `afun(x)` returns `A*x`.

"Parameterizing Functions" explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `minres` converges, a message to that effect is displayed. If `minres` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`minres(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `minres` uses the default, `1e-6`.

`minres(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `minres` uses the default, `min(n,20)`.

`minres(A,b,tol,maxit,M)` and `minres(A,b,tol,maxit,M1,M2)` use symmetric positive definite preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(sqrt(M))*A*inv(sqrt(M))*y =`

inv(sqrt(M))*b for y and then return x = inv(sqrt(M))*y. If M is [] then minres applies no preconditioner. M can be a function handle mfun, such that mfun(x) returns M\x.

minres(A,b,tol,maxit,M1,M2,x0) specifies the initial guess. If x0 is [], then minres uses the default, an all-zero vector.

[x,flag] = minres(A,b,...) also returns a convergence flag.

Flag	Convergence
0	minres converged to the desired tolerance tol within maxit iterations.
1	minres iterated maxit times but did not converge.
2	Preconditioner M was ill-conditioned.
3	minres stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during minres became too small or too large to continue computing.

Whenever flag is not 0, the solution x returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the flag output is specified.

[x,flag,relres] = minres(A,b,...) also returns the relative residual norm(b-A*x)/norm(b). If flag is 0, relres <= tol.

[x,flag,relres,iter] = minres(A,b,...) also returns the iteration number at which x was computed, where 0 <= iter <= maxit.

[x,flag,relres,iter,resvec] = minres(A,b,...) also returns a vector of estimates of the minres residual norms at each iteration, including norm(b-A*x0).

[x,flag,relres,iter,resvec,resveccg] = minres(A,b,...) also returns a vector of estimates of the Conjugate Gradients residual norms at each iteration.

Examples

Using minres with a Matrix Input

```
n = 100; on = ones(n,1);
A = spdiags([-2*on 4*on -2*on],-1:1,n,n);
b = sum(A,2);
tol = 1e-10;
maxit = 50;
M1 = spdiags(4*on,0,n,n);

x = minres(A,b,tol,maxit,M1);
minres converged at iteration 49 to a solution with relative
residual 4.7e-014
```

Using minres with a Function Handle

This example replaces the matrix A in the previous example with a handle to a matrix-vector product function afun. The example is contained in a file run_minres that

- Calls minres with the function handle @afun as its first argument.
- Contains afun as a nested function, so that all variables in run_minres are available to afun.

The following shows the code for run_minres:

```
function x1 = run_minres
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -2*on],-1:1,n,n);
b = sum(A,2);
tol = 1e-10;
maxit = 50;
M = spdiags(4*on,0,n,n);
x1 = minres(@afun,b,tol,maxit,M);

       function y = afun(x)
          y = 4 * x;
          y(2:n) = y(2:n) - 2 * x(1:n-1);
          y(1:n-1) = y(1:n-1) - 2 * x(2:n);
```

```
        end
end
```

When you enter

```
x1=run_minres;
```

MATLAB software displays the message

```
minres converged at iteration 49 to a solution with relative
residual 4.7e-014
```

Using minres instead of pcg

Use a symmetric indefinite matrix that fails with pcg.

```
A = diag([20:-1:1, -1:-1:-20]);
b = sum(A,2);      % The true solution is the vector of all ones.
x = pcg(A,b);      % Errors out at the first iteration.
```

displays the following message:

```
pcg stopped at iteration 1 without converging to the desired
tolerance 1e-006 because a scalar quantity became too small or
too large to continue computing.
The iterate returned (number 0) has relative residual 1
```

However, minres can handle the indefinite matrix A.

```
x = minres(A,b,1e-6,40);
minres converged at iteration 39 to a solution with relative
residual 1.3e-007
```

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Paige, C. C. and M. A. Saunders, "Solution of Sparse Indefinite Systems of Linear Equations." *SIAM J. Numer. Anal.*, Vol.12, 1975, pp. 617-629.

See Also

bicg | bicgstab | cgs | cholinc | gmres | lsqr | pcg | qmr | symmlq | function_handle | mldivide

Purpose

Minus

Syntax

```
c = a-b
c = minus(a,b)
```

Description

`c = a-b` subtracts array `b` from array `a` and returns the result in `c`. Inputs `a` and `b` must have the same size unless one is a scalar value (1-by-1 matrix).

`c = minus(a,b)` is called for the syntax `a-b` when `a` or `b` is an object.

See Also

`plus`

mislocked

Purpose

Determine if function is locked in memory

Syntax

```
mislocked
mislocked(fun)
```

Description

`mislocked` by itself returns logical `1` (`true`) if the currently running function is locked in memory, and logical `0` (`false`) otherwise. Functions that are locked cannot be removed with the `clear` function unless you first unlock them using the `munlock` function. You can use locking on functions that reside in MATLAB `.m` files or `.mex` files.

`mislocked(`*`fun`*`)` returns logical `1` (`true`) if the function named *`fun`* is locked in memory, and logical `0` (`false`) otherwise.

See Also

`mlock` | `munlock` | `inmem`

Purpose

Make new folder

Alternatives

As an alternative to `mkdir`, use the Current Folder browser.

Syntax

```
mkdir('folderName')
mkdir('parentFolder','folderName')
status = mkdir(...)
[status,message,messageid] = mkdir(...)
```

Description

`mkdir('folderName')` creates the folder `folderName`, where `folderName` can be an absolute or a relative path.

`mkdir('parentFolder','folderName')` creates the folder `folderName` in `parentFolder`, where `parentFolder` is an absolute or relative path. If `parentFolder` does not exist, MATLAB attempts to create it. See the Tips section.

`status = mkdir(...)` creates the specified folder. When the operation is successful, it returns a `status` of logical 1. When the operation is unsuccessful, it returns logical 0.

`[status,message,messageid] = mkdir(...)` creates the specified folder, and returns the status, message string, and MATLAB message ID. The value given to `status` is logical `1` for success, and logical `0` for error.

Tips

If an argument specifies a path that includes one or more nonexistent folders, MATLAB attempts to create the nonexistent folder. For example, for

```
mkdir('myFolder\folder1\folder2\targetFolder')
```

if `folder1` does not exist, MATLAB creates `folder1`, creates `folder2` within `folder1`, and creates `targetFolder` within `folder2`.

Examples

Creating a Subfolder in the Current Folder

Create a subfolder called `newdir` in the current folder:

```
mkdir('newdir')
```

Creating a Subfolder in the Specified Parent Folder

Create a subfolder called `newFolder` in the folder `testdata`, using a relative path, where `newFolder` is at the same level as the current folder:

```
mkdir('../testdata','newFolder')
```

Returning Status When Creating a Folder

In this example, the first attempt to create `newFolder` succeeds, returning a status of `1`, and no error or warning message or message identifier:

```
[s, mess, messid] = mkdir('../testdata', 'newFolder')
s =
   1
mess =
   ''
messid =
   ''
```

Attempt to create the same folder again. `mkdir` again returns a success status, and also a warning and message identifier informing you that the folder exists:

```
[s,mess,messid] = mkdir('../testdata','newFolder')
s =
   1
mess =
   Directory "newFolder" already exists.
messid =
   MATLAB:MKDIR:DirectoryExists
```

See Also

`copyfile` | `cd` | `dir` | `ls` | `movefile` | `rmdir`

How To

- “Working with Files and Folders”

Purpose

Create folder on FTP server

Syntax

`mkdir(ftpobj,folder)`

Description

`mkdir(ftpobj,folder)` creates the specified folder on the FTP server associated with `ftpobj`.

Input Arguments

ftpobj

FTP object created by `ftp`.

folder

String enclosed in single quotation marks that specifies a path relative to the current folder on the FTP server.

Examples

Suppose that a hypothetical host, `ftp.testsite.com`, contains a folder named `testfolder`. Connect to the server and add a subfolder:

```
test=ftp('ftp.testsite.com');
mkdir(test,'testfolder/newfolder');
close(test);
```

See Also

`dir` | `ftp` | `rmdir`

mkpp

Purpose

Make piecewise polynomial

Syntax

```
pp = mkpp(breaks,coefs)
pp = mkpp(breaks,coefs,d)
```

Description

`pp = mkpp(breaks,coefs)` builds a piecewise polynomial `pp` from its breaks and coefficients. `breaks` is a vector of length `L+1` with strictly increasing elements which represent the start and end of each of `L` intervals. `coefs` is an `L`-by-`k` matrix with each row `coefs(i,:)` containing the coefficients of the terms, from highest to lowest exponent, of the order `k` polynomial on the interval `[breaks(i),breaks(i+1)]`.

`pp = mkpp(breaks,coefs,d)` indicates that the piecewise polynomial `pp` is `d`-vector valued, i.e., the value of each of its coefficients is a vector of length `d`. `breaks` is an increasing vector of length `L+1`. `coefs` is a `d`-by-`L`-by-`k` array with `coefs(r,i,:)` containing the `k` coefficients of the `i`th polynomial piece of the `r`th component of the piecewise polynomial.

Use `ppval` to evaluate the piecewise polynomial at specific points. Use `unmkpp` to extract details of the piecewise polynomial.

Note. The *order* of a polynomial tells you the number of coefficients used in its description. A *k*th order polynomial has the form

$$c_1x^{k-1} + c_2x^{k-2} + \ldots + c_{k-1}x + c_k$$

It has k coefficients, some of which can be 0, and maximum exponent $k - 1$. So the order of a polynomial is usually one greater than its degree. For example, a cubic polynomial is of order 4.

Examples

The first plot shows the quadratic polynomial

$$1 - \left(\frac{x}{2} - 1\right)^2 = \frac{-x^2}{4} + x$$

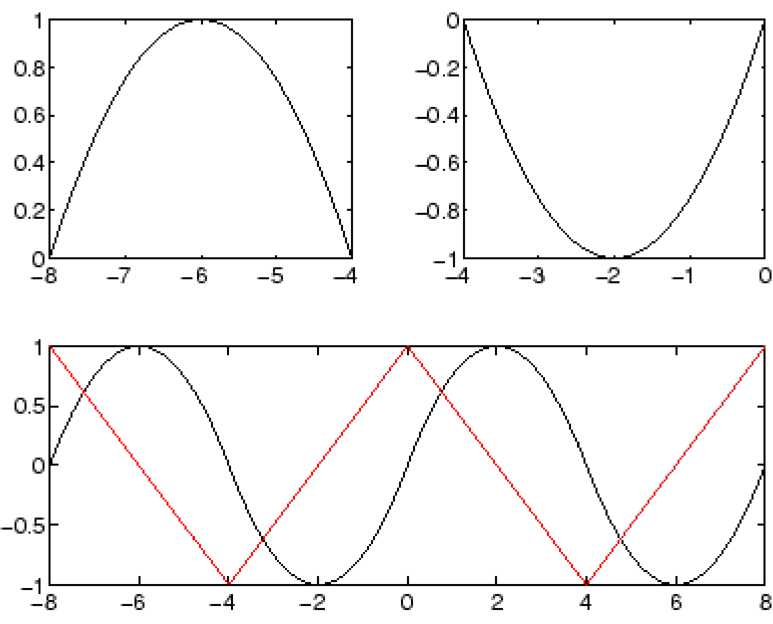
shifted to the interval [-8,-4]. The second plot shows its negative

$$\left(\frac{x}{2}-1\right)^2-1=\frac{x^2}{4}-x$$

but shifted to the interval [-4,0].

The last plot shows a piecewise polynomial constructed by alternating these two quadratic pieces over four intervals. It also shows its first derivative, which was constructed after breaking the piecewise polynomial apart using unmkpp.

```
subplot(2,2,1)
cc = [-1/4 1 0];
pp1 = mkpp([-8 -4],cc);
xx1 = -8:0.1:-4;
plot(xx1,ppval(pp1,xx1),'k-')

subplot(2,2,2)
pp2 = mkpp([-4 0],-cc);
xx2 = -4:0.1:0;
plot(xx2,ppval(pp2,xx2),'k-')

subplot(2,1,2)
pp = mkpp([-8 -4 0 4 8],[cc;-cc;cc;-cc]);
xx = -8:0.1:8;
plot(xx,ppval(pp,xx),'k-')
[breaks,coefs,l,k,d] = unmkpp(pp);
dpp = mkpp(breaks,repmat(k-1:-1:1,d*l,1).*coefs(:,1:k-1),d);
hold on, plot(xx,ppval(dpp,xx),'r-'), hold off
```



See Also

`ppval` | `spline` | `unmkpp`

Purpose

Solve systems of linear equations $Ax = B$ for x

Syntax

```
x = A\B
x = mldivide(A,B)
```

Description

`x = A\B` solves the system of linear equations `A*x = B`. The matrices `A` and `B` must have the same number of rows. MATLAB displays a warning message if `A` is badly scaled or nearly singular, but performs the calculation regardless.

- If `A` is a scalar, `A\B` performs element-wise division of `A` into `B`.
- If `A` is a square `n`-by-`n` matrix and `B` is a column vector with `n` elements or a matrix with `n` rows, then `x = A\B` is a solution to the equation `A*x = B`, if it exists.
- If `A` is a rectangular `m`-by-`n` matrix with `m ~= n`, and `B` is a column vector with `m` elements or a matrix with `m` rows, then `A\B` returns a least-squares solution to the system of equations `A*x= B`.

`x = mldivide(A,B)` is an alternative way to execute `x = A\B`, but is rarely used. It enables operator overloading for classes.

Input Arguments

A - Coefficient matrix

vector | full matrix | sparse matrix

Coefficient matrix, specified as a vector, full matrix, or sparse matrix. If `A` has `m` rows, then `B` must have `m` rows.

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

B - Right-hand side

vector | full matrix | sparse matrix

Right-hand side, specified as a vector, full matrix, or sparse matrix. If `B` has `m` rows, then `A` must have `m` rows.

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

Output Arguments

x - Solution

vector | full matrix | sparse matrix

Solution, specified as a vector, full matrix, or sparse matrix. If A is an m-by-n matrix and B is an m-by-p matrix, then x is an n-by-p matrix, including the case when p==1.

x is sparse only if both A and B are sparse matrices.

Examples

System of Equations

Solve a simple system of linear equations, A*x = B.

```
A = magic(3);
B = [15; 15; 15];
x = A\B
```

```
x =

    1.0000
    1.0000
    1.0000
```

Linear System with Singular Matrix

Solve a linear system of equations involving a singular matrix, C.

```
C = magic(4);
D = [34; 34; 34; 34];
x = C\D
```

```
Warning: Matrix is close to singular or badly scaled. Results may be inac
1.306145e-17.

x =
```

```
   1.5000
   2.5000
  -0.5000
   0.5000
```

MATLAB issues a warning but proceeds with the calculation. MATLAB might give a valid solution even when working with a singular matrix. These solutions are often not unique.

Least-Squares Solution of Underdetermined System

Solve a system of linear equations, `A*x = b`.

```
A = [1 2 0; 0 4 3];
b = [8; 18];
x = A\b

ans =

        0
   4.0000
   0.6667
```

Linear System with Sparse Matrix

Solve a simple system of linear equations using sparse matrices.

Consider the matrix equation `A*x = B`.

```
A = sparse([0 2 0 1 0; 4 -1 -1 0 0; 0 0 0 3 -6; -2 0 0 0 2; 0 0 4 2 0]
B = sparse([8; -1; -18; 8; 20]);
x = A\B

x =

  (1,1)        1.0000
  (2,1)        2.0000
  (3,1)        3.0000
  (4,1)        4.0000
```

```
(5,1)        5.0000
```

Tips

- If A is a square matrix, A\B is roughly equal to inv(A)*B, but MATLAB processes A\B differently and more robustly.
- If the rank of A is less than the number of columns in A, then x = A\B is not necessarily the minimum norm solution. The more computationally expensive x = pinv(A)*B computes the minimum norm least-squares solution.
- For full singular inputs, you can compute the least-squares solution using the function linsolve.

See Also

mrdivide | ldivide | rdivide | inv | pinv | chol | lu | qr | linsolve | ldl

Concepts

- "Systems of Linear Equations"
- "Implementing Operators for Your Class"

Purpose

Solve systems of linear equations $xA = B$ for x

Syntax

```
x = B/A
x = mrdivide(B,A)
```

Description

`x = B/A` solves the system of linear equations `x*A = B` for `x`. The matrices `A` and `B` must contain the same number of columns. MATLAB displays a warning message if `A` is badly scaled or nearly singular, but performs the calculation regardless.

- If `A` is a scalar, `B/A` performs element-wise division of `B` by `A`.
- If `A` is a square `n`-by-`n` matrix and `B` is a row vector with `n` elements or a matrix with `n` columns, then `x = B/A` is a solution to the equation `x*A = B`, if it exists.
- If `A` is a rectangular `m`-by-`n` matrix with `m ~= n`, and `B` is a row vector with `n` elements or a matrix with `n` columns, then `x = B/A` returns a least-squares solution of the system of equations `x*A = B`.

`x = mrdivide(B,A)` is an alternative way to execute `x = B/A`, but is rarely used. It enables operator overloading for classes.

Input Arguments

A - Coefficient matrix

vector | full matrix | sparse matrix

Coefficient matrix, specified as a vector, full matrix, or sparse matrix. If `A` has `n` columns, then `B` must have `n` columns.

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

B - Right-hand side

vector | full matrix | sparse matrix

Right-hand side, specified as a vector, full matrix, or sparse matrix. If `B` has `n` columns, then `A` must have `n` columns.

Data Types
single | double
Complex Number Support: Yes

Output Arguments

x - Solution

vector | full matrix | sparse matrix

Solution, specified as a vector, full matrix, or sparse matrix. If A is an m-by-n matrix and B is a p-by-n matrix, then x is a p-by-m matrix.

x is sparse only if both A and B are sparse matrices.

Examples

System of Equations

Solve a system of equations that has a unique solution, x*A = B.

```
A = [1 1 3; 2 0 4; -1 6 -1];
B = [2 19 8];
x = B/A

x =

    1.0000    2.0000    3.0000
```

Least-Squares on an Underdetermined System

Solve an underdetermined system, x*C = D.

```
C = [1 0; 2 0; 1 0];
D = [1 2];
x = D/C

Warning: Rank deficient, rank = 1, tol =  6.280370e-16.

x =

         0    0.5000         0
```

MATLAB issues a warning but proceeds with calculation.

Verify that `x` is not an exact solution.

```
x*C-D
```

```
ans =

     0    -2
```

Tips

- The operators `/` and `\` are related to each other by the equation `B/A = (A'\B')'`.
- If `A` is a square matrix, `B/A` is roughly equal to `B*inv(A)`, but MATLAB processes `B/A` differently and more robustly.

See Also

`mldivide | ldivide | rdivide | inv | transpose`

Concepts

- "Systems of Linear Equations"

mlint

Purpose

Check MATLAB code files for possible problems

Note `mlint` is not recommended. Use `checkcode` instead.

Alternatives

For information on using the graphical user interface to the Code Analyzer, see "Check Code for Errors and Warnings".

Syntax

```
mlint('filename')
mlint('filename','-config=settings.txt')
mlint('filename','-config=factory')
inform=mlint('filename','-struct')
msg=mlint('filename','-string')
[inform,filepaths]=mlint('filename')
inform=mlint('filename','-id')
inform=mlint('filename','-fullpath')
inform=mlint('filename','-notok')
mlint('filename','-cyc')
mlint('filename','-codegen')
mlint('filename','-eml')
```

Description

`mlint('filename')` displays Code Analyzer messages about `filename`, where the message reports potential problems and opportunities for code improvement. The line number in the message is a hyperlink that opens the file in the Editor, scrolled to that line. If `filename` is a cell array, information is displayed for each file. For `mlint(F1,F2,F3,...)`, where each input is a character array, MATLAB software displays information about each input file name. You cannot combine cell arrays and character arrays of file names. Note that the exact text of the `mlint` messages is subject to some change between versions.

`mlint('filename','-config=settings.txt')` overrides the default active settings file with the settings that enable or suppress messages as indicated in the specified `settings.txt` file.

Note If used, you must specify the full path to the `settings.txt` file specified with the `-config` option.

For information about creating a `settings.txt` file, see “Save and Reuse Code Analyzer Message Settings”. If you specify an invalid file, `mlint` returns a message indicating that it cannot open or read the file you specified. In that case, `mlint` uses the factory default settings.

`mlint('filename','-config=factory')` ignores all settings files and uses the factory default preference settings.

`inform=mlint('filename','-struct')` returns the information in a structure array whose length is the number of messages found. The structure has the fields that follow.

Field	Description
`message`	Message describing the suspicious construct that code analysis caught.
`line`	Vector of file line numbers to which the message refers.
`column`	Two-column array of file columns (column extents) to which the message applies. The first column of the array specifies the column in the Editor where the message begins. The second column of the array specifies the column in the Editor where the message ends. There is one row in the two-column array for each occurrence of a message.

If you specify multiple file names as input, or if you specify a cell array as input, `inform` contains a cell array of structures.

`msg=mlint('filename','-string')` returns the information as a string to the variable `msg`. If you specify multiple file names as input, or if you specify a cell array as input, `msg` contains a string where each

file's information is separated by 10 equal sign characters (=), a space, the file name, a space, and 10 equal sign characters.

If you omit the **`-struct`** or **`-string`** argument and you specify an output argument, the default behavior is **`-struct`**. If you omit the argument and there are no output arguments, the default behavior is to display the information to the command line.

`[inform,filepaths]=mlint('filename')` additionally returns `filepaths`, the absolute paths to the file names, in the same order as you specified them.

`inform=mlint('filename','-id')` requests the message ID, where ID is a string of the form `ABC...`. When returned to a structure, the output also has the `id` field, which is the ID associated with the message.

`inform=mlint('filename','-fullpath')` assumes that the input file names are absolute paths, so that `mlint` does not try to locate them.

`inform=mlint('filename','-notok')` runs `mlint` for all lines in `filename`, even those lines that end with the `mlint` suppression directive, `%#ok`.

`mlint('filename','-cyc')` displays the McCabe complexity (also referred to as cyclomatic complexity) of each function in the file. Higher McCabe complexity values indicate higher complexity, and there is some evidence to suggest that programs with higher complexity values are more likely to contain errors. Frequently, you can lower the complexity of a function by dividing it into smaller, simpler functions. In general, smaller complexity values indicate programs that are easier to understand and modify. Some people advocate splitting up programs that have a complexity rating over 10.

`mlint('filename','-codegen')` enables code generation messages for display in the Command Window.

`mlint('filename','-eml')` `'-eml'` is not recommended. Use `'-codegen'` instead.

Examples

The following examples use `lengthofline.m`, which is a sample file with MATLAB code that can be improved. You can find it in

matlabroot/help/techdoc/matlab_env/examples. If you want to run the examples, save a copy of lengthofline.m to a location on your MATLAB path.

Running mlint on a File with No Options

To run mlint on the example file, lengthofline.m, run

```
mlint('lengthofline')
```

MATLAB displays the M-Lint messages for lengthofline.m in the Command Window:

```
L 22 (C 1-9): The value assigned here to variable 'nothandle' might never be used.
L 23 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 24 (C 5-11): 'notline' might be growing inside a loop. Consider preallocating for speed.
L 24 (C 44-49): Use STRCMPI(str1,str2) instead of using LOWER in a call to STRCMP.
L 28 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 34 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 34 (C 24-31): Use dynamic fieldnames with structures instead of GETFIELD.
                Type 'doc struct' for more information.
L 38 (C 29): Use || instead of | as the OR operator in (scalar) conditional statements.
L 39 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 40 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 42 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 43 (C 13-15): 'dim' might be growing inside a loop. Consider preallocating for speed.
L 45 (C 13-15): 'dim' might be growing inside a loop.Consider preallocating for speed.
L 48 (C 52): There may be a parenthesis imbalance around here.
L 48 (C 53): There may be a parenthesis imbalance around here.
L 48 (C 54): There may be a parenthesis imbalance around here.
L 48 (C 55): There may be a parenthesis imbalance around here.
L 49 (C 17): Terminate statement with semicolon to suppress output (in functions).
L 49 (C 23): Use of brackets [] is unnecessary. Use parentheses to group, if needed.
```

For details about these messages and how to improve the code, see “Changing Code Based on Code Analyzer Messages” in the MATLAB Desktop Tools and Development Environment documentation.

Running mlint with Options to Show IDs and Return Results to a Structure

To store the results to a structure and include message IDs, run

```
inform=mlint('lengthofline', '-id')
```

MATLAB returns

```
inform =

19x1 struct array with fields:
    message
    line
    column
    id
```

To see values for the first message, run

```
inform(1)
```

MATLAB displays

```
ans =

    message: 'The value assigned here to variable 'nothandle' might never be used.'
       line: 22
     column: [1 9]
         id: 'NASGU'
```

Here, the message is for the value that appears on line 22 that extends from column 1–9 in the file.`NASGU` is the ID for the message `'The value assigned here to variable 'nothandle' might never be used.'`.

Displaying McCabe Complexity with mlint

To display the McCabe complexity of a MATLAB code file, run `mlint` with the `-cyc` option, as shown in the following example (assuming you have saved `lengthofline.m` to a local folder).

```
mlint lengthofline.m -cyc
```

Results displayed in the Command Window show the McCabe complexity of the file, followed by the M-Lint messages, as shown here:

```
L 1 (C 23-34): The McCabe complexity of 'lengthofline' is 12.
L 22 (C 1-9): The value assigned here to variable 'nothandle' might never be used.
L 23 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 24 (C 5-11): 'notline' might be growing inside a loop. Consider preallocating for speed.
L 24 (C 44-49): Use STRCMPI(str1,str2) instead of using UPPER/LOWER in a call to STRCMP.
L 28 (C 12-15): NUMEL(x) is usually faster than PROD(SIZE(x)).
L 34 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 34 (C 24-31): Use dynamic fieldnames with structures instead of GETFIELD. Type 'doc struct' for mo
L 38 (C 29): Use || instead of | as the OR operator in (scalar) conditional statements.
L 39 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 40 (C 47): Use || instead of | as the OR operator in (scalar) conditional statements.
L 42 (C 13-16): 'data' might be growing inside a loop. Consider preallocating for speed.
L 43 (C 13-15): 'dim' might be growing inside a loop. Consider preallocating for speed.
L 45 (C 13-15): 'dim' might be growing inside a loop. Consider preallocating for speed.
L 48 (C 52): There may be a parenthesis imbalance around here.
L 48 (C 53): There may be a parenthesis imbalance around here.
L 48 (C 54): There may be a parenthesis imbalance around here.
L 48 (C 55): There may be a parenthesis imbalance around here.
L 49 (C 17): Terminate statement with semicolon to suppress output (in functions).
L 49 (C 23): Use of brackets [] is unnecessary.  Use parentheses to group, if needed.
```

See Also

`mlintrpt`, `profile`

How To

- For information on the suppression directive, `%#ok`, and suppressing messages from within your program, see “Adjust Code Analyzer Message Indicators and Messages”.

mlintrpt

Purpose

Run `checkcode` for file or folder, reporting results in browser

Syntax

```
mlintrpt
mlintrpt('filename','file')
mlintrpt('dirname','dir')
mlintrpt('filename','file','settings.txt')
mlintrpt('dirname','dir','settings.txt')
```

Description

`mlintrpt` scans all files with an `.m` file extension in the current folder for Code Analyzer messages and reports the results in a MATLAB Web browser.

`mlintrpt('filename','file')` scans `filename` for Code Analyzer messages and reports results. You can omit **`'file'`** in this form of the syntax because it is the default.

`mlintrpt('dirname','dir')` scans the specified folder. Here, `dirname` can be in the current folder or can be a full path.

`mlintrpt('filename','file','settings.txt')` applies the Code Analyzer preference settings to enable or suppress messages as indicated in the specified `settings.txt` file.

`mlintrpt('dirname','dir','settings.txt')` applies the settings indicated in the specified `settings.txt` file.

Note If you specify a `settings.txt` file, you must specify the full path to the file.

Examples

`lengthofline.m` is an example file with code that can be improved. It is found in *`matlabroot`*`/matlab/help/techdoc/matlab_env/examples`.

Run Report for All Files in a Folder

Run

```
mlintrpt(fullfile(matlabroot,'help','techdoc','matlab_env','examples'),'dir')
```

and MATLAB displays a report of potential problems and improvements for all files with an `.m` file extension in the `examples` folder.

For details about these messages and how to improve the code, see "Changing Code Based on Code Analyzer Messages".

Run Report Using Code Analyzer Preference Settings

You can save preference settings to a text file by clicking the **Preferences** button in the **Environment** section on the **Home** tab and selecting **Code Analyzer** in the left pane. To save a preferences file, select **Save as** under the drop-down list. To apply those settings when you run `mlintrpt`, use the `file` option and supply the full path to the settings file name as shown in this example:

```
mlintrpt('lengthofline.m', 'file', ...
'C:\WINNT\Profiles\me\Application Data\MathWorks\MATLAB\R2012b\mymlint.txt')
```

Alternatively, use `fullfile` if the settings file is stored in the preferences folder:

```
mlintrpt('lengthofline.m', 'file', fullfile(prefdir,'mymlint.txt'))
```

Assuming that in that example `mymlint.txt` file, the setting for `Terminate statement with semicolon to suppress output` has been disabled, the results of `mlintrpt` for `lengthofline` do not show that message for line 49.

When `mlintrpt` cannot locate the settings file, the first message in the report is

```
0: Unable to open or read the configuration file 'mymlint.txt'--using default settings.
```

See Also

`checkcode`

Concepts

- "Check Code for Errors and Warnings"

mlock

Purpose

Prevent clearing function from memory

Syntax

```
mlock
```

Description

`mlock` locks the currently running function in memory so that subsequent `clear` functions do not remove it. Locking a function in memory also prevents any `persistent` variables defined in the file from getting reinitialized.

Use the `munlock` function to return the file to its normal, clearable state.

Examples

The function `testfun` begins with an `mlock` statement.

```
function testfun
mlock
   .
   .
```

When you execute this function, it becomes locked in memory. You can check this using the `mislocked` function.

```
testfun

mislocked('testfun')
ans =
     1
```

Using `munlock`, you unlock the `testfun` function in memory. Checking its status with `mislocked` shows that it is indeed unlocked at this point.

```
munlock('testfun')

mislocked('testfun')
ans =
     0
```

See Also

`mislocked` | `munlock` | `inmem` | `persistent`

Purpose

Information about multimedia file

Syntax

`info = mmfileinfo(filename)`

Description

info = mmfileinfo(*filename*) returns a structure, *info*, with fields containing information about the contents of the multimedia file identified by *filename*. The *filename* input is a string enclosed in single quotation marks.

If *filename* is a URL, `mmfileinfo` might take a long time to return because it must first download the file. For large files, downloading can take several minutes. To avoid blocking the MATLAB command line while this processing takes place, download the file before calling `mmfileinfo`.

The *info* structure contains the following fields, listed in the order they appear in the structure.

Field	Description
`Filename`	String indicating the name of the file.
`Path`	String indicating the absolute path to the file.
`Duration`	Length of the file in seconds.
`Audio`	Structure containing information about the audio data in the file. See "Audio Data" on page 1-3378 for more information about this data structure.
`Video`	Structure containing information about the video data in the file. See "Video Data" on page 1-3378 for more information about this data structure.

Audio Data

The `Audio` structure contains the following fields, listed in the order they appear in the structure. If the file does not contain audio data, the fields in the structure are empty.

Field	Description
`Format`	Text string, indicating the audio format.
`NumberOfChannels`	Number of audio channels.

Video Data

The `Video` structure contains the following fields, listed in the order they appear in the structure. If the file does not contain video data, the fields in the structure are empty.

Field	Description
`Format`	Text string, indicating the video format.
`Height`	Height of the video frame.
`Width`	Width of the video frame.

Examples

Display information about the example file `xylophone.mpg`:

```
info = mmfileinfo('xylophone.mpg')
audio = info.Audio
video = info.Video
```

MATLAB returns:

```
info =
    Filename: 'xylophone.mpg'
        Path: [1x75 char]
    Duration: 4.7020
       Audio: [1x1 struct]
       Video: [1x1 struct]
```

```
audio =
              Format: 'MPEG'
    NumberOfChannels: 2

video =
    Format: 'MPEG1'
    Height: 240
     Width: 320
```

where `Path` is system-dependent.

See Also

`get` | `VideoReader`

mmreader

Purpose

Create object for reading video files

Note `mmreader` will be removed in a future release. Use `VideoReader` instead. The `mmreader` class has been renamed `VideoReader`.

Description

Use `mmreader` with the `read` method to read video data from a multimedia file into the MATLAB workspace.

The file formats that `mmreader` supports vary by platform, as follows (with no restrictions on file extensions):

All Platforms	Motion JPEG 2000 (`.mj2`)
Windows	AVI (`.avi`), MPEG-1 (`.mpg`), Windows Media Video (`.wmv`, `.asf`, `.asx`), and any format supported by Microsoft DirectShow.
Macintosh	AVI (`.avi`), MPEG-1 (`.mpg`), MPEG-4 (`.mp4`, `.m4v`), Apple QuickTime Movie (`.mov`), and any format supported by QuickTime as listed on http://support.apple.com/kb/HT3775.
Linux	Any format supported by your installed plug-ins for GStreamer 0.10 or above, as listed on http://gstreamer.freedesktop.org/documentation/plugins.html, including AVI (`.avi`) and Ogg Theora (`.ogg`).

For more information, see “Supported Video File Formats” in the MATLAB Data Import and Export documentation.

Construction

`obj = mmreader(filename)` constructs `obj` to read video data from the file named `filename`. The `mmreader` constructor searches for the file on the MATLAB path. If it cannot construct the object for any reason, `mmreader` generates an error.

`obj = mmreader(filename,'PropertyName',PropertyValue)` constructs the object using options, specified as property name/value pairs. Property name/value pairs can be in any format that the `set` method supports: name/value string pairs, structures, or name/value cell array pairs.

Properties

BitsPerPixel

Bits per pixel of the video data. (Read-only)

Duration

Total length of the file in seconds. (Read-only)

FrameRate

Frame rate of the video in frames per second. (Read-only)

Height

Height of the video frame in pixels. (Read-only)

Name

Name of the file associated with the object. (Read-only)

NumberOfFrames

Total number of frames in the video stream. (Read-only)

Some files store video at a variable frame rate, including many Windows Media Video files. For these files, `mmreader` cannot determine the number of frames until you read the last frame. When you construct the object, `mmreader` returns a warning and does not set the `NumberOfFrames` property.

To count the number of frames in a variable frame rate file, use the `read` method to read the last frame of the file. For example:

```
vidObj = mmreader('varFrameRateFile.wmv');
lastFrame = read(vidObj, inf);
numFrames = vidObj.NumberOfFrames;
```

For more information, see “Reading Variable Frame Rate Video” in the MATLAB Data Import and Export documentation.

Path

String containing the full path to the file associated with the reader. (Read-only)

Tag

User-defined string to identify the object.

Default: `''`

Type

Class name of the object: `'mmreader'`. (Read-only)

UserData

Generic field for user-defined data.

Default: `[]`

VideoFormat

String indicating the MATLAB representation of the video format, such as `'RGB24'`. (Read-only)

Width

Width of the video frame in pixels. (Read-only)

Methods

For backward compatibility, `mmreader` supports the following `VideoReader` methods:

get	Query property values for video reader object
getFileFormats	File formats that `VideoReader` supports

read	Read video frame data from file
set	Set property values for video reader object

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Construct an `mmreader` object for the example movie file `xylophone.mpg` and view its properties:

```
xyloObj = mmreader('xylophone.mpg', 'Tag', 'My reader object');
get(xyloObj)
```

Read and play back the movie file `xylophone.mpg`:

```
xyloObj = mmreader('xylophone.mpg');

nFrames = xyloObj.NumberOfFrames;
vidHeight = xyloObj.Height;
vidWidth = xyloObj.Width;

% Preallocate movie structure.
mov(1:nFrames) = ...
    struct('cdata', zeros(vidHeight, vidWidth, 3, 'uint8'),...
           'colormap', []);

% Read one frame at a time.
for k = 1 : nFrames
    mov(k).cdata = read(xyloObj, k);
end

% Size a figure based on the video's width and height.
hf = figure;
set(hf, 'position', [150 150 vidWidth vidHeight])
```

```
% Play back the movie once at the video's frame rate.
movie(hf, mov, 1, xyloObj.FrameRate);
```

See Also

mmfileinfo | VideoReader

How To

- “Read Video Files”

Purpose

Modulus after division

Syntax

```
M = mod(X,Y)
```

Description

`M = mod(X,Y)` if `Y ~= 0`, returns `X - n.*Y` where `n = floor(X./Y)`. If `Y` is not an integer and the quotient `X./Y` is within roundoff error of an integer, then `n` is that integer. The inputs `X` and `Y` must be real arrays of the same size, or real scalars.

The following are true by convention:

- `mod(X,0)` is `X`
- `mod(X,X)` is `0`
- `mod(X,Y)` for `X~=Y` and `Y~=0` has the same sign as `Y`.

Tips

`rem(X,Y)` for `X~=Y` and `Y~=0` has the same sign as `X`.

`mod(X,Y)` and `rem(X,Y)` are equal if `X` and `Y` have the same sign, but differ by `Y` if `X` and `Y` have different signs.

The `mod` function is useful for congruence relationships: *x and y are congruent (mod m)* if and only if `mod(x,m) == mod(y,m)`.

Examples

```
mod(13,5)
ans =
     3

mod([1:5],3)
ans =
     1     2     0     1     2

mod(magic(3),3)
ans =
     2     1     0
     0     2     1
     1     0     2
```

See Also

`rem`

Purpose

Most frequent values in array

Syntax

```
M = mode(X)
M = mode(X,dim)

[M,F] = mode( ___ )
[M,F,C] = mode( ___ )
```

Description

`M = mode(X)` returns the sample mode of `X`, which is the most frequently occurring value in `X`. When there are multiple values occurring equally frequently, `mode` returns the smallest of those values. For complex inputs, the smallest value is the first value in a sorted list.

- If `X` is a vector, then `mode(X)` returns the most frequent value of `X`.
- If `X` is a nonempty matrix, then `mode(X)` returns a row vector containing the mode of each column of `X`.
- If `X` is an empty 0-by-0 matrix, `mode(X)` returns `NaN`.
- If `X` is a multidimensional array, then `mode(X)` acts along the first nonsingleton dimension and returns an array of most frequent values. The size of this dimension reduces to `1` while the sizes of all other dimensions remain the same.

`M = mode(X,dim)` returns the mode of elements along dimension `dim`. For example, if `X` is a matrix, then `mode(X,2)` is a column vector containing the most frequent value of each row

`[M,F] = mode( ___ )` also returns a frequency array `F`, using any of the input arguments in the previous syntaxes. `F` is the same size as `M`, and each element of `F` represents the number of occurrences of the corresponding element of `M`.

`[M,F,C] = mode( ___ )` also returns a cell array `C` of the same size as `M` and `F`. Each element of `C` is a sorted vector of all values that have the same frequency as the corresponding element of `M`.

Input Arguments

X - Input array

vector | matrix | multidimensional array

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

`NaN` values in the input array, `X`, are ignored.

Data Types

`double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

dim - Dimension to operate along

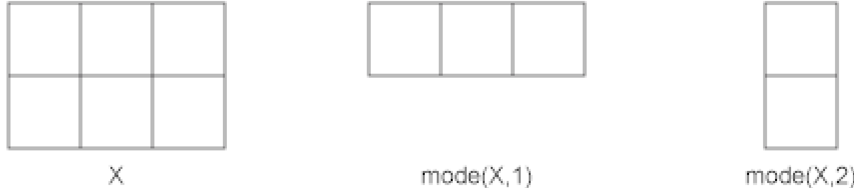
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, the default is the first nonsingleton dimension.

Dimension `dim` indicates the dimension whose length reduces to `1`. The `size(M,dim)` is `1`, while the sizes of all other dimensions remain the same.

Consider a two-dimensional input array, `X`.

- If `dim = 1`, then `mode(X,1)` returns a row vector containing the most frequent value in each column.
- If `dim = 2`, then `mode(X,2)` returns a column vector containing the most frequent value in each row.



`mode` returns `X` if `dim` is greater than `ndims(X)`.

Data Types

`double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

M - Most frequent values

scalar | vector | matrix | multidimensional array

Most frequent values returned as a scalar, vector, matrix, or multidimensional array. When there are multiple values occurring equally frequently, `mode` returns the smallest of those values. For complex inputs, this is taken to be the first value in a sorted list of values.

The class of `M` is the same as the class of the input array, `X`.

F - Frequency array

scalar | vector | matrix | multidimensional array

Frequency array returned as a scalar, vector, matrix, or multidimensional array. The size of `F` is the same as the size of `M`, and each element of `F` represents the number of occurrences of the corresponding element of `M`.

The class of `F` is always `double`.

C - Most frequent values with multiplicity

cell array

Most frequent values with multiplicity returned as a cell array. The size of `C` is the same as the size of `M` and `F`, and each element of `C` is a sorted column vector of all values that have the same frequency as the corresponding element of `M`.

Definitions

First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to `1`.

For example:

- If `X` is a 1-by-n row vector, then the second dimension is the first nonsingleton dimension of `X`.
- If `X` is a 1-by-0-by-n empty array, then the second dimension is the first nonsingleton dimension of `X`.

- If X is a 1-by-1-by-3 array, then the third dimension is the first nonsingleton dimension of X.

Tips

- The mode function is most useful with discrete or coarsely rounded data. The mode for a continuous probability distribution is defined as the peak of its density function. Applying the mode function to a sample from that distribution is unlikely to provide a good estimate of the peak; it would be better to compute a histogram or density estimate and calculate the peak of that estimate. Also, the mode function is not suitable for finding peaks in distributions having multiple modes.

Examples

Mode of Matrix Columns

Define a 3-by-4 matrix.

```
X = [3 3 1 4; 0 0 1 1; 0 1 2 4]
```

```
X =

     3     3     1     4
     0     0     1     1
     0     1     2     4
```

Find the most frequent value of each column.

```
M = mode(X)
```

```
M =

     0     0     1     4
```

Mode of Matrix Rows

Define a 3-by-4 matrix.

```
X = [3 3 1 4; 0 0 1 1; 0 1 2 4]
```

```
X =

     3     3     1     4
     0     0     1     1
     0     1     2     4
```

Find the most frequent value of each row.

```
M = mode(X,2)
```

```
M =

     3
     0
     0
```

Mode of 3-D Array

Create a 1-by-3-by-4 array of integers between 1 and 10.

```
X = gallery('integerdata',10,[1,3,4],1)
```

```
X(:,:,1) =

    10     8    10


X(:,:,2) =

     6     9     5


X(:,:,3) =

     9     6     1


X(:,:,4) =
```

```
     4     9     5
```

Find the most frequent values of this 3-D array along the second dimension.

```
M = mode(X)
```

```
M(:,:,1) =

    10


M(:,:,2) =

     5


M(:,:,3) =

     1


M(:,:,4) =

     4
```

This operation produces a 1-by-1-by-4 array by finding the most frequent value along the second dimension. The size of the second dimension reduces to `1`.

Compute the mode along the first dimension of `X`.

```
M = mode(X,1);
isequal(X,M)
```

```
ans =
```

```
     1
```

This returns the same array as `X` because the size of the first dimension is `1`.

Mode of Matrix Columns with Frequency Information

Define a 3-by-4 matrix.

```
X = [3 3 1 4; 0 0 1 1; 0 1 2 4]
```

```
X =

     3     3     1     4
     0     0     1     1
     0     1     2     4
```

Find the most frequent value of each column, as well as how often it occurs.

```
[M,F] = mode(X)
```

```
M =

     0     0     1     4


F =

     2     1     2     2
```

`F(1)` is `2` since `M(1)` occurs twice in the first column.

Mode of Matrix Rows with Frequency and Multiplicity Information

Define a 3-by-4 matrix.

```
X = [3 3 1 4; 0 0 1 1; 0 1 2 4]
```

```
X =

     3     3     1     4
     0     0     1     1
     0     1     2     4
```

Find the most frequent value of each row, how often it occurs, and which values in that row occur with the same frequency.

```
[M,F,C] = mode(X,2)

M =

     3
     0
     0


F =

     2
     2
     1


C =

    [         3]
    [2x1 double]
    [4x1 double]
```

`C{2}` is the 2-by-1 vector `[0;1]` since values `0` and `1` in the second row occur with frequency `F(2)`.

`C{3}` is the 4-by-1 vector `[0;1;2;4]` since all values in the third row occur with frequency `F(3)`.

Mode of 16-bit Unsigned Integer Array

Define a 1-by-4 vector of 16-bit unsigned integers.

```
X = gallery('integerdata',10,[1,4],3,'uint16')
```

```
X =

      6      3      2      3
```

Find the most frequent value, as well as the number of times it occurs.

```
[M,F] = mode(X),
class(M)
```

```
M =

      3


F =

     2


ans =

uint16
```

M is the same class as the input, X.

See Also

mean | median | hist | histc | sort

more

Purpose

Control paged output for Command Window

Syntax

```
more on
more off
more(n)
A = more(state)
```

Description

`more on` enables paging of the output in the MATLAB Command Window. MATLAB displays output one page at a time. Use the keys defined in the table below to control paging.

`more off` disables paging of the output in the MATLAB Command Window.

`more(n)` defines the length of a page to be `n` lines.

`A = more(state)` returns in `A` the number of lines that are currently defined to be a page. The `state` input can be one of the quoted strings `'on'` or `'off'`, or the number of lines to set as the new page length.

By default, the length of a page is equal to the number of lines available for display in the MATLAB Command Window. Manually changing the size of the command window adjusts the page length accordingly.

If you set the page length to a specific value, MATLAB uses that value for the page size, regardless of the size of the command window. To have MATLAB return to matching page size to window size, type `more off` followed by `more on`.

To see the status of `more`, type `get(0,'More')`. MATLAB returns either `on` or `off`, indicating the `more` status.

When you have enabled `more` and are examining output, you can do the following.

Press the...	To...
Return key	Advance to the next line of output.
Space bar	Advance to the next page of output.
Q (for quit) key	Terminate display of the text. Do not use **Ctrl+C** to terminate `more` or you might generate error messages in the Command Window.

`more` is in the **`off`** state, by default.

See Also

`diary`

move

Purpose

Move or resize control in parent window

Syntax

```
V = h.move(position)
V = move(h, position)
```

Description

`V = h.move(position)` moves the control to the position specified by the `position` argument. When you use move with only the handle argument, `h`, it returns a four-element vector indicating the current position of the control.

`V = move(h, position)` is an alternate syntax.

The position argument is a four-element vector specifying the position and size of the control in the parent figure window. The elements of the vector are:

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

where `x` and `y` are offsets, in pixels, from the bottom left corner of the figure window to the same corner of the control, and `width` and `height` are the size of the control itself.

Examples

This example moves the control:

```
f = figure('Position', [100 100 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.1', [0 0 200 200], f);
pos = h.move([50 50 200 200])
pos =
    50    50   200   200
```

The next example resizes the control to always be centered in the figure as you resize the figure window. Start by creating the script `resizectrl.m` that contains

```
% Get the new position and size of the figure window
  fpos = get(gcbo, 'position');

% Resize the control accordingly
```

```
  h.move([0 0 fpos(3) fpos(4)]);
```

Now execute the following:

```
f = figure('Position', [100 100 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.1', [0 0 200 200]);
set(f, 'ResizeFcn', 'resizectrl');
```

As you resize the figure window, notice that the circle moves so that it is always positioned in the center of the window.

See Also

set (COM) | get (COM)

movefile

Purpose

Move file or folder

Alternatives

As an alternative to the `movefile` function, use the Current Folder browser. For more information, see “Working with Files and Folders”.

Syntax

```
movefile('source')
movefile('source','destination')
movefile('source','destination','f')
[status,message,messageid]=movefile(...)
```

Description

`movefile('source')` moves the file or folder named `source` to the current folder, where `source` is the absolute or relative path name for the folder or file. To move multiple files or folders, use one or more wildcard characters (`*`) after the last file separator in `source`. The `source` argument permits a wildcard character in a path string. `movefile` does not preserve the archive attribute of `source`.

`movefile('source','destination')` moves the file or folder named `source` to the location `destination`, where `source` and `destination` are the absolute or relative paths for the folder or file. To move multiple files or folders, you can use one or more wildcard characters (`*`) after the last file separator in `source`. You cannot use a wildcard character in `destination`. To rename a file or folder when moving it, make `destination` a different name than `source`, and specify only one file for `source`. When `source` and `destination` have the same location, `movefile` renames `source` to `destination`.

`movefile('source','destination','f')` moves the file or folder named `source` to the location `destination`, regardless of the read-only attribute of `destination`.

`[status,message,messageid]=movefile(...)` moves the file or folder named `source` to the location `destination`, returning the status, a message, and the MATLAB message ID. Here, `status` is logical `1` for success or logical `0` for error. `movefile` requires only one output argument.

Examples

Moving a File to the Current Folder

Assuming `myfiles` is a subfolder within the current folder, move the file `myfunction.m` to the current folder:

```
movefile('myfiles/myfunction.m')
```

Assuming `projects/testcases` is the current folder, move `projects/myfiles` and its contents to the current folder:

```
movefile('../myfiles')
```

Renaming a File in the Current Folder

In the current folder, rename `myfunction.m` to `oldfunction.m`:

```
movefile('myfunction.m','oldfunction.m')
```

Using a Wildcard to Move All Matching Files

Assuming `myfiles` is a subfolder of the current folder, move all files whose names that begin with `my` from the `myfiles` folder, to the current folder:

```
movefile('myfiles/my*')
```

Moving a File to a Different Folder

Assuming `projects` and the current folder are at the same level, move the file `myfunction.m` from the current folder to the folder `projects`:

```
movefile('myfunction.m','../projects')
```

Moving a Folder Down One Level

Assuming `projects` is a subfolder of the current folder, move the folder `projects/testcases` and all its contents down a level in `projects` into `projects/myfiles`:

```
movefile('projects/testcases','projects/myfiles/')
```

Moving a File to a Read-Only Folder and Renaming the File

Move the file `myfile.m` from the current folder to `d:/work/restricted`, assigning it the name `test1.m`, where `restricted` is a read-only folder:

```
movefile('myfile.m','d:/work/restricted/test1.m','f')
```

The read-only file `myfile.m` is no longer in the current folder. The file `test1.m` is in `d:/work/restricted` and is read only.

Returning Status When Moving Files

Move all files in the folder `myfiles` whose names start with `new` to the current folder, when there is an error. You mistype `new*` as `nex*` and no items in the current folder start with `nex*`:

```
[s,mess,messid]=movefile('myfiles/nex*')

s =
     0

mess =

No matching files were found.

messid =

MATLAB:MOVEFILE:FileDoesNotExist
```

See Also

`cd` | `copyfile` | `delete` | `dir` | `fileattrib` | `ls` | `mkdir` | `rmdir`

How To

- “Creating, Opening, Changing, and Deleting Files and Folders”

Purpose

Move GUI figure to specified location on screen

Syntax

```
movegui(h,'position')
movegui(position)
movegui(h)
movegui
```

Description

`movegui(h,'position')` moves the figure identified by handle `h` to the specified screen location, preserving the figure's size. The *`position`* argument is a string or a two-element vector, as defined in the following tables.

`movegui(position)` moves the callback figure (`gcbf`) or the current figure (`gcf`) to the specified position.

`movegui(h)` moves the figure identified by the handle `h` to the `onscreen` position.

`movegui` moves the callback figure (`gcbf`) or the current figure (`gcf`) to the `onscreen` position. You can specify `'movegui'` as a `CreateFcn` callback for a figure. It ensures after you save it, the figure appears on screen when you reload it, regardless of its saved position. See the following example.

When it is a string, *`position`* is one of the following descriptors.

Position String	Description
`north`	Top center edge of screen
`south`	Bottom center edge of screen
`east`	Right center edge of screen
`west`	Left center edge of screen
`northeast`	Top right corner of screen
`northwest`	Top left corner of screen
`southeast`	Bottom right corner of screen

Position String	Description
`southwest`	Bottom left corner
`center`	Centered on screen
`onscreen`	Nearest location to current location that is entirely on screen

You can also specify the *`position`* argument as a two-element vector, `[h,v]`. Depending on sign, `h` specifies the figure's offset from the left or right edge of the screen, and `v` specifies the figure's offset from the top or bottom of the screen, in pixels. The following table summarizes the possible values.

`h` (for `h >= 0`)	Offset of left side from left edge of screen
`h` (for `h < 0`)	Offset of right side from right edge of screen
`v` (for `v >= 0`)	Offset of bottom edge from bottom of screen
`v` (for `v < 0`)	Offset of top edge from top of screen

When you apply `movegui` to a maximized figure window, the window can shrink in size by a few pixels. On Microsoft Windows platforms, using it to move a maximized window toward the Windows task bar creates a gap on the opposite side of the screen about as wide as the task bar.

`GUIDE` and `openfig` call `movegui` when loading figures to ensure they are visible.

Examples

Use `movegui` to ensure that a saved GUI appears on screen you reload it, regardless of the target computer screen size and resolution. Create a figure that is off the screen, assign `movegui` as its `CreateFcn` callback, save it, and then reload the figure.

```
f = figure('Position',[10000,10000,400,300]);
```

```
% The figure does not display because it is created offscreen
set(f,'CreateFcn','movegui')
hgsave(f,'onscreenfig')
close(f)
f2 = hgload('onscreenfig');
% The reloaded figure is now visible
```

See Also

guide | openfig

Tutorials

- “About the Simple Programmatic GUI”
- “GUI for Manipulating Data That Persists Across MATLAB Sessions (GUIDE)”

How To

- “Positioning Figures”

movie

Purpose

Play recorded movie frames

Syntax

```
movie(M)
movie(M,n)
movie(M,n,fps)
movie(h,...)
movie(h,M,n,fps,loc)
```

Description

The `movie` function plays the movie defined by a matrix whose columns are movie frames (usually produced by `getframe`).

`movie(M)` plays the movie in matrix `M` once, using the current axes as the default target. If you want to play the movie in the figure instead of the axes, specify the figure handle (or `gcf`) as the first argument: `movie(`*`figure_handle`*`,...)`. `M` must be an array of movie frames (usually from `getframe`).

`movie(M,n)` plays the movie `n` times. If `n` is negative, each cycle is shown forward then backward. If `n` is a vector, the first element is the number of times to play the movie, and the remaining elements make up a list of frames to play in the movie.

For example, if `M` has four frames then `n = [10 4 4 2 1]` plays the movie ten times, and the movie consists of frame `4` followed by frame `4` again, followed by frame `2` and finally frame `1`.

`movie(M,n,fps)` plays the movie at `fps` frames per second. The default is 12 frames per second. Computers that cannot achieve the specified speed play as fast as possible.

`movie(h,...)` plays the movie centered in the figure or axes identified by the handle `h`. Specifying the figure or axes enables MATLAB to fit the movie to the available size.

`movie(h,M,n,fps,loc)` specifies `loc`, a four-element location vector, `[x y 0 0]`, where the lower left corner of the movie frame is anchored (only the first two elements in the vector are used). The location is relative to the lower left corner of the figure or axes specified by `handle h` and in units of pixels, regardless of the object's `Units` property.

Tips

The `movie` function uses a default figure size of 560-by-420 and does not resize figures to fit movies with larger or smaller frames. To accommodate other frame sizes, you can resize the figure to fit the movie, as shown in the second example below.

`movie` only accepts 8-bit image frames; it does not accept 16-bit grayscale or 24–bit truecolor image frames.

Buffering the movie places all frames in memory. As a result, on Microsoft Windows and perhaps other platforms, a long movie (on the order of several hundred frames) can exhaust memory, depending on system resources. In such cases an error message is issued that says

```
??? Error using ==> movie
Could not create movie frame
```

You can abort a movie by typing **Ctrl-C**.

`movie` is not a built-in function. Therefore, you cannot call `movie` using the `builtin` function.

Limitations with Renderer on Windows Systems

Setting the figure `Renderer` property to `zbuffer` or `painters` works around limitations of using `getframe` with the `OpenGL` renderer on some Windows systems.

Examples

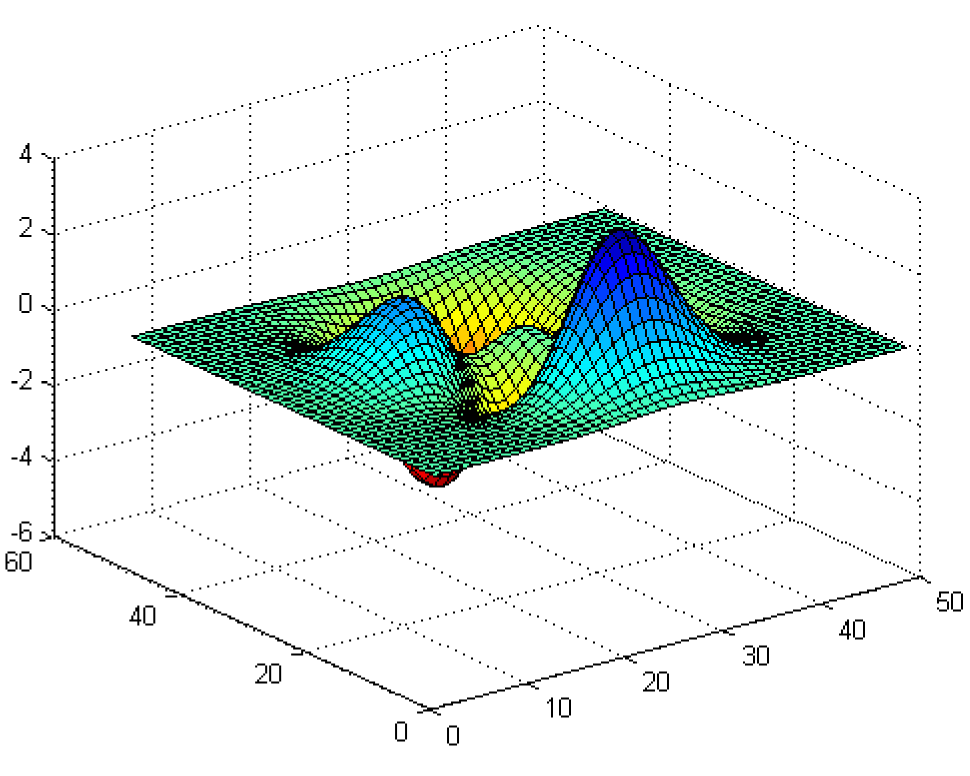
Animate the `peaks` function as you scale the values of `Z`:

```
figure('Renderer','zbuffer')
Z = peaks;
surf(Z);
axis tight
set(gca,'NextPlot','replaceChildren');
% Preallocate the struct array for the struct returned by getframe
F(20) = struct('cdata',[],'colormap',[]);
% Record the movie
for j = 1:20
    surf(.01+sin(2*pi*j/20)*Z,Z)
    F(j) = getframe;
```

```
end
```

Now play the movie ten times. The twelfth frame looks like the following plot.

```
movie(F,10)
```



With larger frames, first adjust the figure's size to fit the movie:

```
figure('Position',[100 100 850 600])
```

```
Z = peaks; surf(Z);
axis tight
set(gca,'NextPlot','replacechildren');
% Record the movie
for j = 1:20
    surf(sin(2*pi*j/20)*Z,Z)
    F(j) = getframe;
end
% use 1st frame to get dimensions
[h, w, p] = size(F(1).cdata);
hf = figure;
% resize figure based on frame's w x h, and place at (150, 150)
set(hf,'Position', [150 150 w h]);
axis off
% Place frames at bottom left
movie(hf,F,4,30,[0 0 0 0]);
```

See Also

`getframe` | `frame2im` | `im2frame` | `VideoReader` | `VideoWriter`

How To

- Example – Visualizing an FFT as a Movie

Purpose

Create Audio/Video Interleaved (AVI) file from MATLAB movie

Syntax

```
movie2avi(mov, filename)
movie2avi(mov, filename, ParameterName, ParameterValue)
```

Description

`movie2avi(mov, filename)` creates the AVI file *filename* from the MATLAB movie *mov*. The *filename* input is a string. The *mov* input is a 1-by-*n* structure array, where *n* is the number of frames. Each frame is a structure with two fields: `cdata` and `colormap`. For more information, see `getframe`.

`movie2avi(mov, filename, ParameterName, ParameterValue)` accepts one or more comma-separated parameter name/value pairs. The following table lists the available parameters and values.

Parameter Name	Value	Default
`'colormap'`	An *m*-by-3 matrix defining the colormap for indexed AVI movies, where *m* is no more than 256 (236 for `Indeo` compression). Valid only when the `'compression'` is `'MSVC'`, `'RLE'`, or `'None'`.	No default
`'compression'`	A text string specifying the compression codec to use. To create an uncompressed file, specify a value of `'None'`. On UNIX operating systems, the only valid value is `'None'`. On Windows systems, valid values include: • `'MSVC'` • `'RLE'` • `'Cinepak'` on 32-bit systems.	`'Indeo5'` on Windows systems. `'None'` on UNIX systems.

Parameter Name	Value	Default
	• 'Indeo3' or 'Indeo5' on 32-bit Windows XP systems. Alternatively, specify a custom compression codec on Windows systems using the four-character code that identifies the codec (typically included in the codec documentation). If MATLAB cannot find the specified codec, it returns an error.	
'fps'	A scalar value specifying the speed of the AVI movie in frames per second (fps).	15 fps
'keyframe'	For compressors that support temporal compression, the number of key frames per second.	2.1429 key frames per second
'quality'	A number from 0 through 100. Higher quality numbers result in higher video quality and larger file sizes. Lower quality numbers result in lower video quality and smaller file sizes. Valid only for compressed movies.	75
'videoname'	A descriptive name for the video stream, no more than 64 characters.	*filename*

Tips

- On some Windows systems, including all 64-bit systems, the default Indeo 5 codec is not available. MATLAB issues a warning, and creates an uncompressed file.
- On 32-bit Windows XP systems, MATLAB can create AVI files compressed with Indeo 3 and Indeo 5 codecs. However, Microsoft Windows XP Service Pack 3 (SP3) with Security Update 954157 disables playback of Indeo 3 and Indeo 5 codecs in Windows Media Player and Internet Explorer. Consider specifying a `compression` value of `'None'`.

Examples

Create a movie and write to an uncompressed AVI file, `myPeaks.avi`:

```
nFrames = 20;

% Preallocate movie structure.
mov(1:nFrames) = struct('cdata', [],...
                        'colormap', []);

% Create movie.
Z = peaks; surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
for k = 1:nFrames
   surf(sin(2*pi*k/20)*Z,Z)
   mov(k) = getframe(gcf);
end

% Create AVI file.
movie2avi(mov, 'myPeaks.avi', 'compression', 'None');
```

See Also

`VideoWriter` | `VideoReader` | `mmfileinfo` | `movie`

Purpose

Matrix power

Syntax

```
c = a^b
c = mpower(a,b)
```

Description

`c = a^b` computes `a` to the `b` power and returns the result in `c`. Inputs `a` and `b` must be one of the following:

- `a` is a square matrix and exponent `b` a scalar.
- `a` is a scalar and exponent `b` a square matrix.

`c = mpower(a,b)` is called for the syntax `a^b` when `a` or `b` is an object.

See Also

`power`

FTP.mput

Purpose

Upload file or folder to FTP server

Syntax

```
mput(ftpobj,contents)
paths = mput(ftpobj,contents)
```

Description

`mput(ftpobj,contents)` uploads the file or folder specified by `contents` to the current folder on an FTP server.

`paths = mput(ftpobj,contents)` returns a cell array that lists the paths to the uploaded files on the server.

Input Arguments

ftpobj

FTP object created by `ftp`.

contents

String enclosed in single quotation marks that specifies either a file name or a folder name. Can include a wildcard character (`*`).

Output Arguments

paths

Cell array that includes the paths to the uploaded files on the server.

Examples

Suppose that your current MATLAB folder contains files `myfile1.m` through `myfile10.m`, and that you want to upload to a hypothetical FTP server, `ftp.testsite.com`. Connect to the server and upload the files:

```
test = ftp('ftp.testsite.com');
mput(test, 'myfile*.m');
close(test);
```

See Also

`mget` | `ftp` | `mkdir` | `rename`

Purpose

Create and open message box

Syntax

```
h = msgbox(Message)
h = msgbox(Message,Title)
h = msgbox(Message,Title,Icon)
h = msgbox(Message,Title,'custom',IconData,IconCMap)
h = msgbox(...,CreateMode)
```

Description

`h = msgbox(Message)` creates a message dialog box that automatically wraps `Message` to fit an appropriately sized figure. `Message` is a string vector, string matrix, or cell array. `msgbox` returns the handle of the message box in `h`.

`h = msgbox(Message,Title)` specifies the title of the message box.

`h = msgbox(Message,Title,Icon)` specifies which icon to display in the message box. `Icon` is `'none'`, `'error'`, `'help'`, `'warn'`, or `'custom'`. The default is `'none'`.

Error Icon

Help Icon

Warning Icon

`h = msgbox(Message,Title,'custom',IconData,IconCMap)` defines a customized icon. `IconData` contains image data defining the icon. `IconCMap` is the colormap used for the image.

`h = msgbox(...,CreateMode)` specifies whether the message box is modal or nonmodal. Optionally, it can also specify an interpreter for `Message` and `Title`.

If *`CreateMode`* is a string, it must be one of the values shown in the following table.

CreateMode Value	Description
`'modal'`	Replaces the message box having the specified `Title`, that was last created or clicked on, with a modal message box as specified. All other message boxes with the same title are deleted. The message box which is replaced can be either modal or nonmodal.
`'non-modal'` (default)	Creates a new nonmodal message box with the specified parameters. Existing message boxes with the same title are not deleted.
`'replace'`	Replaces the message box having the specified `Title`, that was last created or clicked on, with a nonmodal message box as specified. All other message boxes with the same title are deleted. The message box which is replaced can be either modal or nonmodal.

Note A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution as well, use the `uiwait` function.

If you open a dialog with `errordlg`, `msgbox`, or `warndlg` using `'CreateMode','modal'` and a non-modal dialog created with any of these functions is already present and *has the same name as the modal dialog*, the non-modal dialog closes when the modal one opens.

For more information about modal dialog boxes, see `WindowStyle` in the Figure Properties.

If *`CreateMode`* is a structure, it can have fields `WindowStyle` and `Interpreter`. The `WindowStyle` field must be one of the values in the

table above. `Interpreter` is one of the strings`'tex'` or `'none'`. The default value for `Interpreter` is `'none'`.

See Also

`dialog` | `errordlg` | `helpdlg` | `inputdlg` | `listdlg` | `questdlg` | `warndlg` | `figure` | `textwrap` | `uiwait` | `uiresume`

mtimes

Purpose

Matrix multiplication

Syntax

```
C = A*B
```

Description

`C = A*B` is the linear algebraic product of the matrices `A` and `B`. If `A` is an m-by-p and `B` is a p-by-n matrix, the `i,j` entry of `C` is defined by

$$C(i,j) = \sum_{k=1}^{p} A(i,k)B(k,j).$$

The product `C` is an m-by-n matrix. For nonscalar `A` and `B`, the number of columns of `A` must equal the number of rows of `B`. You can multiply a scalar by a matrix of any size.

The preceding definition says that `C(i,j)` is the inner product of the `i`th row of `A` with the `j`th column of `B`. You can write this definition using the MATLAB colon operator as

```
C(i,j) = A(i,:)*B(:,j)
```

where `A(i,:)` is the `i`th row of `A` and `B(:,j)` is the `j`th column of `B`.

Note If `A` is an m-by-0 empty matrix and `B` is a 0-by-n empty matrix, where m and n are positive integers, `A*B` is an m-by-n matrix of zeros.

Examples

Example 1

If `A` is a row vector and `B` is a column vector with the same number of elements as `A`, `A*B` is the inner product of `A` and `B`. For example,

```
A = [5 3 2 6]

A =

     5     3     2     6
```

```
B = [-4 9 0 1]'

B =

    -4
     9
     0
     1
A*B

ans =

    13
```

Example 2

```
A = [1 3 5; 2 4 7]

A =

     1     3     5
     2     4     7
B = [-5 8 11; 3 9 21;4 0 8]

B =

    -5     8    11
     3     9    21
     4     0     8
```

The product of A and B is

```
C = A*B

C =

    24    35   114
    30    52   162
```

Note that the second row of A is

```
A(2,:)

ans =

     2     4     7
```

while the third column of B is

```
B(:,3)

ans =

    11
    21
     8
```

The inner product of `A(2,:)` and `B(:,3)` is

```
A(2,:)*B(:,3)

ans =

   162
```

which is the same as `C(2,3)`.

See Also Arithmetic Operators

Purpose

Convert mu-law audio signal to linear

Syntax

```
y = mu2lin(mu)
```

Description

`y = mu2lin(mu)` converts mu-law encoded 8-bit audio signals, stored as “flints” in the range `0` ≤ `mu` ≤ `255`, to linear signal amplitude in the range `-s < Y < s` where `s = 32124/32768 ~= .9803`. The input `mu` is often obtained using `fread(...,'uchar')` to read byte-encoded audio files. "Flints" are MATLAB integers — floating-point numbers whose values are integers.

See Also

`auread` | `lin2mu`

multibandread

Purpose

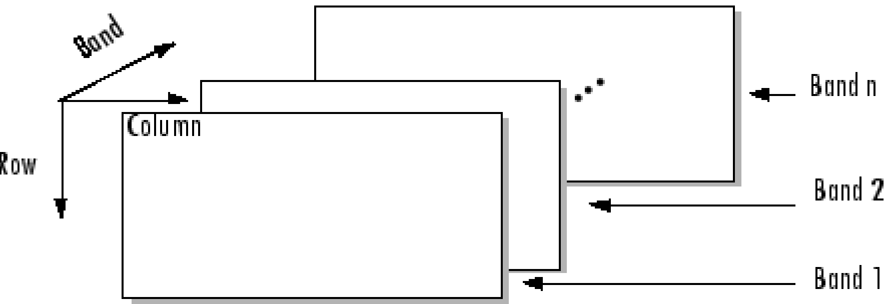
Read band-interleaved data from binary file

Syntax

```
X = multibandread(filename, size, precision, offset,
   interleave,  byteorder)
X = multibandread(...,subset1,subset2,subset3)
```

Description

`X = multibandread(filename, size, precision, offset, interleave, byteorder)` reads band-sequential (BSQ), band-interleaved-by-line (BIL), or band-interleaved-by-pixel (BIP) data from the binary file `filename`. The `filename` input is a string enclosed in single quotes. This function defines *band* as the third dimension in a 3-D array, as shown in this figure.



You can use the parameters to `multibandread` to specify many aspects of the read operation, such as which bands to read. See "Parameters" on page 1-3423 for more information.

`X` is a 2-D array if only one band is read; otherwise it is 3-D. `X` is returned as an array of data type `double` by default. Use the `precision` parameter to map the data to a different data type.

`X = multibandread(...,subset1,subset2,subset3)` reads a subset of the data in the file. You can use up to three subsetting parameters to specify the data subset along row, column, and band dimensions. See "Subsetting Parameters" on page 1-3424 for more information.

Note In addition to BSQ, BIL, and BIP files, multiband imagery may be stored using the TIFF file format. In that case, use the `imread` function to import the data.

Parameters

This table describes the arguments accepted by `multibandread`.

Argument	Description
`filename`	String containing the name of the file to be read.
`size`	Three-element vector of integers consisting of `[height, width, N]`, where • `height` is the total number of rows • `width` is the total number of elements in each row • `N` is the total number of bands. This will be the dimensions of the data if it is read in its entirety.
`precision`	String specifying the format of the data to be read, such as `'uint8'`, `'double'`, `'integer*4'`, or any of the other precisions supported by the `fread` function. Note: You can also use the `precision` parameter to specify the format of the output data. For example, to read `uint8` data and output a `uint8` array, specify a precision of `'uint8=>uint8'` (or `'*uint8'`). To read `uint8` data and output it in the MATLAB software in single precision, specify `'uint8=>single'`. See `fread` for more information.

Argument	Description
`offset`	Scalar specifying the zero-based location of the first data element in the file. This value represents the number of bytes from the beginning of the file to where the data begins.
`interleave`	String specifying the format in which the data is stored • `'bsq'` — Band-Sequential • `'bil'` — Band-Interleaved-by-Line • `'bip'` — Band-Interleaved-by-Pixel For more information about these interleave methods, see the `multibandwrite` reference page.
`byteorder`	String specifying the byte ordering (machine format) in which the data is stored, such as • `'ieee-le'` — Little-endian • `'ieee-be'` — Big-endian See `fopen` for a complete list of supported formats.

Subsetting Parameters

You can specify up to three subsetting parameters. Each subsetting parameter is a three-element cell array, `{`*`dim`*`, `*`method`*`, index}`, where

Parameter	Description
`dim`	Text string specifying the dimension to subset along. It can have any of these values: • `'Column'` • `'Row'`

Parameter	Description
	• 'Band'
method	Text string specifying the subsetting method. It can have either of these values: • 'Direct' • 'Range' If you leave out this element of the subset cell array, multibandread uses 'Direct' as the default.
index	If method is 'Direct', index is a vector specifying the indices to read along the Band dimension. If method is 'Range', index is a three-element vector of [start, increment, stop] specifying the range and step size to read along the dimension specified in dim. If index is a two-element vector, multibandread assumes that the value of increment is 1.

Examples

Example 1

Setup initial parameters for a data set.

```
rows=3; cols=3; bands=5;
filename = tempname;
```

Define the data set.

```
fid = fopen(filename, 'w', 'ieee-le');
fwrite(fid, 1:rows*cols*bands, 'double');
fclose(fid);
```

Read every other band of the data using the Band-Sequential format.

```
im1 = multibandread(filename, [rows cols bands], ...
                    'double', 0, 'bsq', 'ieee-le', ...
                    {'Band', 'Range', [1 2 bands]} )
```

Read the first two rows and columns of data using Band-Interleaved-by-Pixel format.

```
im2 = multibandread(filename, [rows cols bands], ...
                    'double', 0, 'bip', 'ieee-le', ...
                    {'Row', 'Range', [1 2]}, ...
                    {'Column', 'Range', [1 2]} )
```

Read the data using Band-Interleaved-by-Line format.

```
im3 = multibandread(filename, [rows cols bands], ...
                    'double', 0, 'bil', 'ieee-le')
```

Delete the file created in this example.

```
delete(filename);
```

Example 2

Read `int16` BIL data from the FITS file `tst0012.fits`, starting at byte 74880.

```
im4 = multibandread('tst0012.fits', [31 73 5], ...
                    'int16', 74880, 'bil', 'ieee-be', ...
                    {'Band', 'Range', [1 3]} );
im5 = double(im4)/max(max(max(im4)));
imagesc(im5);
```

See Also

`fread` | `fwrite` | `imread` | `memmapfile` | `multibandwrite`

Purpose

Write band-interleaved data to file

Syntax

```
multibandwrite(data,filename,interleave)
multibandwrite(data,filename,interleave,start,totalsize)
multibandwrite(...,param,value...)
```

Description

`multibandwrite(data,filename,interleave)` writes `data`, a two- or three-dimensional numeric or logical array, to the binary file specified by `filename`. The `filename` input is a string enclosed in single quotes. The length of the third dimension of `data` determines the number of bands written to the file. The bands are written to the file in the form specified by `interleave`. See "Interleave Methods" on page 1-3429 for more information about this argument.

If filename already exists, `multibandwrite` overwrites it unless you specify the optional `offset` parameter. See the last alternate syntax for `multibandwrite` for information about other optional parameters.

`multibandwrite(data,filename,interleave,start,totalsize)` writes `data` to the binary file `filename` in chunks. In this syntax, `data` is a subset of the complete data set.

`start` is a 1-by-3 array `[firstrow firstcolumn firstband]` that specifies the location to start writing data. `firstrow` and `firstcolumn` specify the location of the upper left image pixel. `firstband` gives the index of the first band to write. For example, `data(I,J,K)` contains the data for the pixel at `[firstrow+I-1, firstcolumn+J-1]` in the `(firstband+K-1)`-th band.

`totalsize` is a 1-by-3 array, `[totalrows,totalcolumns,totalbands]`, which specifies the full, three-dimensional size of the data to be written to the file.

Note In this syntax, you must call `multibandwrite` multiple times to write all the data to the file. The first time it is called, `multibandwrite` writes the complete file, using the fill value for all values outside the data subset. In each subsequent call, `multibandwrite` overwrites these fill values with the data subset in `data`. The parameters `filename`, `interleave`, `offset`, and `totalsize` must remain constant throughout the writing of the file.

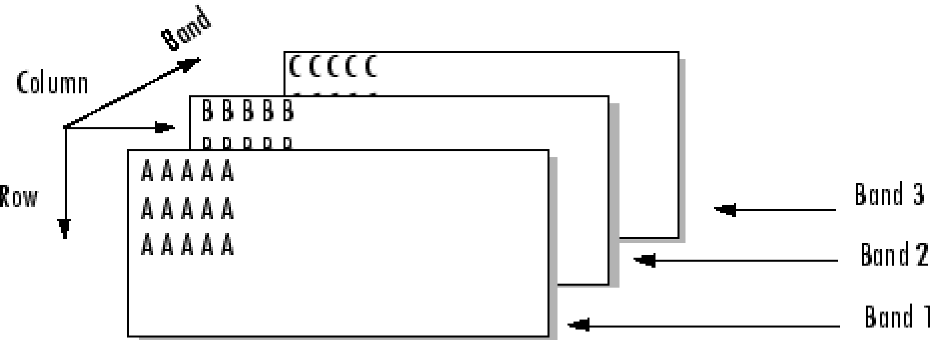
`multibandwrite(...,param,value...)` writes the multiband data to a file, specifying any of these optional parameter/value pairs.

Parameter	Description
'precision'	String specifying the form and size of each element written to the file. See the help for `fwrite` for a list of valid values. The default precision is the class of the data.
'offset'	The number of bytes to skip before the first data element. If the file does not already exist, `multibandwrite` writes ASCII null values to fill the space. To specify a different fill value, use the parameter `'fillvalue'`. This option is useful when you are writing a header to the file before or after writing the data. When writing the header to the file after the data is written, open the file with `fopen` using `'r+'` permission.

Parameter	Description
'machfmt'	String to control the format in which the data is written to the file. Typical values are 'ieee-le' for little endian and 'ieee-be' for big endian. See the help for fopen for a complete list of available formats. The default machine format is the local machine format.
'fillvalue'	A number specifying the value to use in place of missing data. 'fillvalue' can be a single number, specifying the fill value for all missing data, or a 1-by-Number-of-bands vector of numbers specifying the fill value for each band. This value is used to fill space when data is written in chunks.

Interleave Methods

interleave is a string that specifies how multibandwrite interleaves the bands as it writes data to the file. If data is two-dimensional, multibandwrite ignores the interleave argument. The following table lists the supported methods and uses this example multiband file to illustrate each method.



Supported methods of interleaving bands include those listed below.

Method	String	Description	Example
Band-Interleaved-by-Line	'bil'	Write an entire row from each band	AAAAABBBBBCCCCC AAAAABBBBBCCCCC AAAAABBBBBCCCCC
Band-Interleaved-by-Pixel	'bip'	Write a pixel from each band	ABCABCABCABCABC...
Band-Sequential	'bsq'	Write each band in its entirety	AAAAA AAAAA AAAAA BBBBB BBBBB BBBBB CCCCC CCCCC CCCCC

Examples

Note To run these examples successfully, you must be in a writable directory.

Example 1

Write all data (interleaved by line) to the file in one call.

```
data = reshape(uint16(1:600), [10 20 3]);
multibandwrite(data,'data.bil','bil');
```

Example 2

Write a single-band tiled image with one call for each tile. This is only useful if a subset of each band is available at each call to `multibandwrite`.

```
numBands = 1;
dataDims = [1024 1024 numBands];
data = reshape(uint32(1:(1024 * 1024 * numBands)), dataDims);

for band = 1:numBands
   for row = 1:2
      for col = 1:2

         subsetRows = ((row - 1) * 512 + 1):(row * 512);
         subsetCols = ((col - 1) * 512 + 1):(col * 512);

         upperLeft = [subsetRows(1), subsetCols(1), band];
         multibandwrite(data(subsetRows, subsetCols, band), ...
                          'banddata.bsq', 'bsq', upperLeft, dataDims);

      end
   end
end
```

See Also

`multibandread` | `fwrite` | `fread`

munlock

Purpose

Allow clearing functions from memory

Syntax

```
munlock
munlock fun
munlock('fun')
```

Description

`munlock` unlocks the currently running `.m` or `.mex` function in memory so that subsequent `clear` functions can remove it.

`munlock fun` unlocks the `.m` or `.mex` file named `fun` from memory. By default, these files are unlocked so that changes to the file are picked up. Calls to munlock are needed only to unlock `.m` or `.mex` functions that have been locked with `mlock`.

`munlock('fun')` is the function form of `munlock`.

Examples

The function `testfun` begins with an `mlock` statement.

```
function testfun
mlock
   .
   .
```

When you execute this function, it becomes locked in memory. You can check this using the `mislocked` function.

```
testfun

mislocked testfun
ans =
     1
```

Using `munlock`, you unlock the `testfun` function in memory. Checking its status with `mislocked` shows that it is indeed unlocked at this point.

```
munlock testfun

mislocked testfun
ans =
```

```
0
```

See Also

`mlock` | `mislocked` | `inmem` | `persistent`

namelengthmax

Purpose

Maximum identifier length

Syntax

```
len = namelengthmax
```

Description

`len = namelengthmax` returns the maximum length allowed for MATLAB identifiers, which include:

- Variable names
- Structure field names
- Script, function, and class names
- Model names

Rather than hard-coding a specific maximum name length into your programs, use the `namelengthmax` function. This saves you the trouble of having to update these limits should the identifier length change in some future MATLAB release.

Examples

Call `namelengthmax` to get the maximum identifier length:

```
maxid = namelengthmax
maxid =
    63
```

See Also

`isvarname` | `genvarname`

Purpose

Not-a-Number

Syntax

```
NaN
NaN('double')
NaN('single')
NaN(n)
NaN(m,n)
NaN([m,n])
NaN(m,n,p,...)
NaN([m,n,p,...])
NaN(...,'like',p)
```

Description

`NaN` returns the IEEE arithmetic representation for Not-a-Number (`NaN`). These result from operations which have undefined numerical results.

`NaN('double')` is the same as `NaN` with no inputs.

`NaN('single')` is the single precision representation of `NaN`.

`NaN(n)` is an `n`-by-`n` matrix of `NaN`s.

`NaN(m,n)` or `NaN([m,n])` is an `m`-by-`n` matrix of `NaN`s.

`NaN(m,n,p,...)` or `NaN([m,n,p,...])` is an `m`-by-`n`-by-`p`-by-... array of `NaN`s.

Note The size inputs `m`, `n`, `p`, `...` should be nonnegative integers. Negative integers are treated as 0.

`NaN(...,classname)` is an array of `NaN`s of class specified by the string `classname`. `classname` can be either `'single'` or `'double'`.

`NaN(...,'like',p)` is an array of `NaN`s of the same data type, sparsity, and complexity (real or complex) as the single or double precision numeric variable `p`.

Examples

These operations produce NaN:

- Any arithmetic operation on a NaN, such as sqrt(NaN)
- Addition or subtraction, such as magnitude subtraction of infinities as (+Inf)+(-Inf)
- Multiplication, such as 0*Inf
- Division, such as 0/0 and Inf/Inf
- Remainder, such as rem(x,y) where y is zero or x is infinity

Tips

Because two NaNs are not equal to each other, logical operations involving NaNs always return false, except ~= (not equal). Consequently,

```
NaN ~= NaN
ans =
     1
NaN == NaN
ans =
     0
```

and the NaNs in a vector are treated as different unique elements.

```
unique([1 1 NaN NaN])
ans =
     1 NaN NaN
```

Use the isnan function to detect NaNs in an array.

```
isnan([1 1 NaN NaN])
ans =
     0     0     1     1
```

See Also

Inf | isnan

Purpose

Validate number of input arguments

Note `nargchk` will be removed in a future version. Use `narginchk` instead.

Syntax

```
msgstring = nargchk(minargs, maxargs, numargs)
msgstring = nargchk(minargs, maxargs, numargs, 'string')
msgstruct = nargchk(minargs, maxargs, numargs, 'struct')
```

Description

Use `nargchk` inside a function to check that the desired number of input arguments is specified in the call to that function.

`msgstring = nargchk(minargs, maxargs, numargs)` returns an error message string `msgstring` if the number of inputs specified in the call `numargs` is less than `minargs` or greater than `maxargs`. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargchk` returns an empty matrix.

It is common to use the `nargin` function to determine the number of input arguments specified in the call.

`msgstring = nargchk(minargs, maxargs, numargs, 'string')` is essentially the same as the command shown above, as `nargchk` returns a string by default.

`msgstruct = nargchk(minargs, maxargs, numargs, 'struct')` returns an error message structure `msgstruct` instead of a string. The fields of the return structure contain the error message string and a message identifier. If `numargs` is between `minargs` and `maxargs` (inclusive), `nargchk` returns an empty structure.

When too few inputs are supplied, the message string and identifier are

```
       message: 'Not enough input arguments.'
    identifier: 'MATLAB:nargchk:notEnoughInputs'
```

When too many inputs are supplied, the message string and identifier are

```
    message: 'Too many input arguments.'
 identifier: 'MATLAB:nargchk:tooManyInputs'
```

Tips

`nargchk` is often used together with the `error` function. The `error` function accepts either type of return value from `nargchk`: a message string or message structure. For example, this command provides the `error` function with a message string and identifier regarding which error was caught:

```
error(nargchk(2, 4, nargin, 'struct'))
```

If `nargchk` detects no error, it returns an empty string or structure. When `nargchk` is used with the `error` function, as shown here, this empty string or structure is passed as an input to `error`. When `error` receives an empty string or structure, it simply returns and no error is generated.

Examples

Given the function `CheckInputs`,

```
function CheckInputs(x, y, z)
error(nargchk(2, 3, nargin))
```

Then typing `CheckInputs(1)` produces

```
Not enough input arguments.
```

See Also

`narginchk` | `nargoutchk` | `nargin` | `nargout` | `varargin` | `varargout` | `error`

Purpose

Number of function input arguments

Syntax

```
nargin
nargin(fx)
```

Description

`nargin` returns the number of input arguments passed in the call to the currently executing function. Use this `nargin` syntax only in the body of a function.

`nargin(fx)` returns the number of input arguments that appear in the definition statement for function `fx`. If the function includes `varargin` in its definition, then `nargin` returns the negative of the number of inputs. For example, if function `foo` declares inputs `a`, `b`, and `varargin`, then `nargin('foo')` returns `-3`.

Input Arguments

fx

Either a function handle or a string in single quotes that specifies the name of a function.

Examples

Inputs to Current Function

Create a function in a file named `addme.m` that accepts up to two inputs, and identify the number of inputs with `nargin`.

```
function c = addme(a,b)

switch nargin
    case 2
        c = a + b;
    case 1
        c = a + a;
    otherwise
        c = 0;
end
```

Inputs Defined for a Function

Determine how many inputs a function can accept.

The function addme created in the previous example has two inputs in its declaration statement (a and b).

```
fx = 'addme';
nargin(fx)

ans =
     2
```

Function with varargin Input

Determine how many inputs a function that uses varargin can accept.

Define a function in a file named mynewplot.m that accepts numeric inputs x and y and any number of additional plot inputs using varargin.

```
function mynewplot(x,y,varargin)
   figure
   plot(x,y,varargin{:})
   title('My New Plot')
```

At the command line, query how many inputs newplot can accept.

```
fx = 'mynewplot';
nargin(fx)

ans =
    -3
```

The minus sign indicates that the third input is varargin. The mynewplot function can accept an indeterminate number of additional input arguments.

See Also

nargout | narginchk | nargoutchk | varargin | varargout | inputname

Purpose

Validate number of input arguments

Syntax

```
narginchk(minargs, maxargs)
```

Description

`narginchk(minargs, maxargs)` throws an error if the number of inputs specified in the call to the currently executing function is less than `minargs` or greater than `maxargs`. If the number of inputs is between `minargs` and `maxargs` (inclusive), `narginchk` does nothing.

When too few inputs are supplied, the message identifier and message are:

```
    identifier: 'MATLAB:narginchk:notEnoughInputs'
       message: 'Not enough input arguments.'
```

When too many inputs are supplied, the message identifier and message are:

```
    identifier: 'MATLAB:narginchk:tooManyInputs'
       message: 'Too many input arguments.'
```

Examples

This function uses `narginchk` to verify that a minimum of 2 and maximum of 5 input arguments are received from the calling function:

```
function check_inputs(A, B, varargin)
minargs=2;  maxargs=5;

% Number of inputs must be >=minargs and <=maxargs.
narginchk(minargs, maxargs)

fprintf('Received 2 required, %d optional inputs.\n\n', ...
    size(varargin, 2))
```

Call the example function, passing 1 input argument:

```
check_inputs(23)
Error using check_inputs
Not enough input arguments.
```

Call the function, passing 5 arguments:

```
check_inputs(23, 9, 15, 34, 62)
Received 2 required, 3 optional inputs.
```

Call the function, passing 6 arguments:

```
check_inputs(23, 9, 15, 34, 62, 6)
Error using check_inputs
Too many input arguments.
```

See Also

nargoutchk | nargin | nargout | varargin | varargout

Purpose

Number of function output arguments

Syntax

```
nargout
nargout(fx)
```

Description

`nargout` returns the number of output arguments specified in the call to the currently executing function. Use this `nargout` syntax only in the body of a function.

`nargout(fx)` returns the number of outputs that appear in the definition statement of function `fx`. If the function includes `varargout` in its definition, then `nargout` returns the negative of the number of outputs. For example, if function `foo` declares outputs `a`, `b`, and `varargout`, then `nargout('foo')` returns `-3`.

Input Arguments

fx

Either a function handle or a string in single quotes that specifies the name of a function.

Examples

Outputs for Current Function

Create a function in a file named `subtract.m` that calculates a second return value only when requested.

```
function [dif,absdif] = subtract(y,x)
dif = y - x;
if nargout > 1
   disp('Calculating absolute value')
   absdif = abs(dif);
end
```

Outputs Defined for a Function

Determine how many outputs a function can return.

The function named `subtract` created in the previous example has two outputs in its declaration statement (`dif` and `absdif`).

```
fx = 'subtract';
nargout(fx)

ans =
     2
```

Function with varargout Output

Determine how many outputs a function that uses `varargout` can return.

Define a function in a file named `mysize.m` that returns a vector of dimensions from the `size` function and the individual dimensions using `varargout`.

```
function [sizeVector,varargout] = mysize(x)
   sizeVector = size(x);
   varargout = cell(1,nargout-1);
   for k = 1:length(varargout)
      varargout{k} = sizeVector(k);
   end
```

At the command line, query how many outputs `mysize` can return.

```
fx = 'mysize';
nargout(fx)

ans =
    -2
```

The minus sign indicates that the second output is `varargout`. The `mysize` function can return an indeterminate number of additional outputs.

Tips

- When you use a function as part of an expression, MATLAB calls the function with one output argument, so `nargout` within the function returns `1`. For example, given the following `if` statement and the `subtract` function defined in the Examples section, the value of `nargout` within the `subtract` function is `1`.

```
a = 1; b = 2;
if subtract(a,b) < 0
   disp('Result is negative')
end
```

See Also

nargin | nargoutchk | narginchk | varargout | varargin | inputname

nargoutchk

Purpose

Validate number of output arguments

Syntax

```
nargoutchk(minargs, maxargs)
```

Description

`nargoutchk(minargs, maxargs)` throws an error if the number of outputs specified in the call is less than `minargs` or greater than `maxargs`. If the number of outputs is between `minargs` and `maxargs` (inclusive), `nargoutchk` does nothing.

When too few outputs are supplied, the identifier and message are:

```
identifier: 'MATLAB:nargoutchk:notEnoughOutputs'
       message: 'Not enough output arguments.'
```

When too many outputs are supplied, the identifier and message are:

```
identifier: 'MATLAB:nargoutchk:tooManyOutputs'
       message: 'Too many output arguments.'
```

Examples

This function uses `nargoutchk` to verify that a minimum of 2 and maximum of 5 input arguments are passed back to the calling function:

```
function [varargout] = check_outputs(array_in)
minargs=2;  maxargs=5;

% Number of outputs must be >=minargs and <=maxargs.
nargoutchk(minargs, maxargs)

for k=1:nargout
    varargout{k} = array_in(k)*3;
end
```

Initialize input array X to a vector of 6 elements:

```
X = 5:7:40
X =
     5    12    19    26    33    40
```

Call the example function with 1 output argument. This is less than the minimum (2) that was specified by `nargoutchk` and results in an error:

```
A = check_outputs(X);

Error using check_outputs
Not enough output arguments.
```

Call the function with 4 output arguments. This is within the allowable bounds (2 to 5) specified by `nargoutchk`:

```
[A, B, C, D] = check_outputs(X);

[A, B, C, D]
ans =
    15    36    57    78
```

Call the function with 6 output arguments. This exceeds the maximum (5) that was specified by `nargoutchk` and results in an error:

```
[A, B, C, D, E, F] = check_outputs(X);

Error using check_outputs
Too many output arguments.
```

See Also

`narginchk` | `nargout` | `nargin` | `varargout` | `varargin`

native2unicode

Purpose

Convert numeric bytes to Unicode characters

Syntax

```
unicodestr = native2unicode(bytes)
unicodestr = native2unicode(bytes, encoding)
```

Description

`unicodestr = native2unicode(bytes)` takes a vector containing numeric values in the range [0,255] and converts these values as a stream of 8-bit bytes to Unicode characters. The stream of bytes is assumed to be in the MATLAB default character encoding scheme. Return value `unicodestr` is a `char` vector that has the same general array shape as `bytes`.

`unicodestr = native2unicode(bytes, encoding)` does the conversion with the assumption that the byte stream is in the character encoding scheme specified by the string `encoding`. `encoding` must be the empty string (`''`) or a name or alias for an encoding scheme. Some examples are `'UTF-8'`, `'latin1'`, `'US-ASCII'`, and `'Shift_JIS'`. For common names and aliases, see the Web site `http://www.iana.org/assignments/character-sets`. If `encoding` is unspecified or is the empty string (`''`), the MATLAB default encoding scheme is used.

Note If `bytes` is a `char` vector, it is returned unchanged.

Examples

This example begins with a vector of bytes in an unknown character encoding scheme. The user-written function `detect_encoding` determines the encoding scheme. If successful, it returns the encoding scheme name or alias as a string. If unsuccessful, it throws an error represented by an MException object, `ME`. The example calls `native2unicode` to convert the bytes to Unicode characters:

```
try
   enc = detect_encoding(bytes);
    str = native2unicode(bytes, enc);
    disp(str);
```

```
catch ME
    rethrow(ME);
end
```

Note that the computer must be configured to display text in a language represented by the detected encoding scheme for the output of `disp(str)` to be correct.

See Also

`unicode2native`

nchoosek

Purpose

Binomial coefficient or all combinations

Syntax

```
b = nchoosek(n,k)
C = nchoosek(v,k)
```

Description

`b = nchoosek(n,k)` returns the binomial coefficient, defined as $n!/((n-k)!\ k!)$. This is the number of combinations of `n` items taken `k` at a time.

`C = nchoosek(v,k)` returns a matrix containing all possible combinations of the elements of vector `v` taken `k` at a time. Matrix `C` has `k` columns and $n!/((n-k)!\ k!)$ rows, where n is `length(v)`.

Input Arguments

n - Number of possible choices

scalar, real, nonnegative value

Number of possible choices, specified as a scalar value of any numeric type that is real and nonnegative.

Example: `10`

Example: `int16(10)`

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

k - Number of selected choices

scalar, real, nonnegative value

Number of selected choices, specified as a scalar value that is real and nonnegative. `k` can be any numeric type. However, `nchoosek(n,k)` requires that `n` and `k` be the same type or that at least one of them be of type `double`.

There are no restrictions on combining inputs of different types for `nchoosek(v,k)`.

Example: `3`

Example: `int16(3)`

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

v - Set of all choices

vector of numeric, `logical`, or `char` values

Set of all choices, specified as a vector of numeric, `logical`, or `char` values.

Example: `[1 2 3 4 5]`

Example: `[1+1i 2+1i 3+1i 4+1i]`

Example: `int16([1 2 3 4 5])`

Example: `[true false true false]`

Example: `['abcd']`

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

Output Arguments

b - Binomial coefficient

nonnegative scalar value

Binomial coefficient, returned as a nonnegative scalar value. `b` is the same type as `n` and `k`. If `n` and `k` are of different types, then `b` is returned as the nondouble type.

C - All combinations of v

matrix

All combinations of `v`, returned as a matrix of the same type as `v`. Matrix `C` has `k` columns and $n!/((n-k)!\ k!)$ rows, where n is `length(v)`.

Each row of `C` contains a combination of `k` items chosen from `v`. The elements in each row of `C` are listed in the same order as they appear in `v`.

Limitations

- When `b = nchoosek(n,k)` is sufficiently large, `nchoosek` displays a warning that the result might not be exact. In this case, the result is only accurate to 15 digits for double-precision inputs, or 8 digits for single-precision inputs.
- `C = nchoosek(v,k)` is only practical for situations where `length(v)` is less than about `15`.

Examples

Binomial Coefficient, "5 Choose 4"

```
b = nchoosek(5,4)

b =

     5
```

All Combinations of Five Numbers Taken Four at a Time

```
v = 2:2:10;
C = nchoosek(v,4)

C =

     2     4     6     8
     2     4     6    10
     2     4     8    10
     2     6     8    10
     4     6     8    10
```

All Combinations of Three Unsigned Integers Taken Two at a Time

```
v = uint16([10 20 30]);
C = nchoosek(v,uint16(2))
```

```
C =

    10    20
    10    30
    20    30
```

See Also

perms

ndgrid

Purpose

Rectangular grid in N-D space

Syntax

```
[X1,X2,X3,...,Xn] = ndgrid(x1gv,x2gv,x3gv,...,xngv)
[X1,X2,...,Xn] = ndgrid(xgv)
```

Description

`[X1,X2,X3,...,Xn] = ndgrid(x1gv,x2gv,x3gv,...,xngv)` replicates the grid vectors `x1gv`,`x2gv`,`x3gv,...,xngv` to produce a full grid. This grid is represented by the output coordinate arrays `X1`,`X2`,`X3,...,Xn`. The ith dimension of any output array `Xi` contains copies of the grid vector `xigv`.

`[X1,X2,...,Xn] = ndgrid(xgv)` is the same as `[X1,X2,...,Xn] = ndgrid(xgv,xgv,...,xgv)`. In other words, you can reuse the same grid vector in each respective dimension. The dimensionality of the output arrays is determined by the number of output arguments.

The coordinate arrays `[X1,X2,X3,...,Xn]` are typically used to evaluate functions of several variables and to create surface and volumetric plots.

Input Arguments

xigv

Grid vector specifying a series of grid point coordinates in the ith dimension.

xgv

Generic grid vector specifying a series of point coordinates.

Output Arguments

Xi

The `ith` dimension of the output array `Xi` are copies of elements of the grid vector `xigv`. The output arrays specify the full grid.

Tips

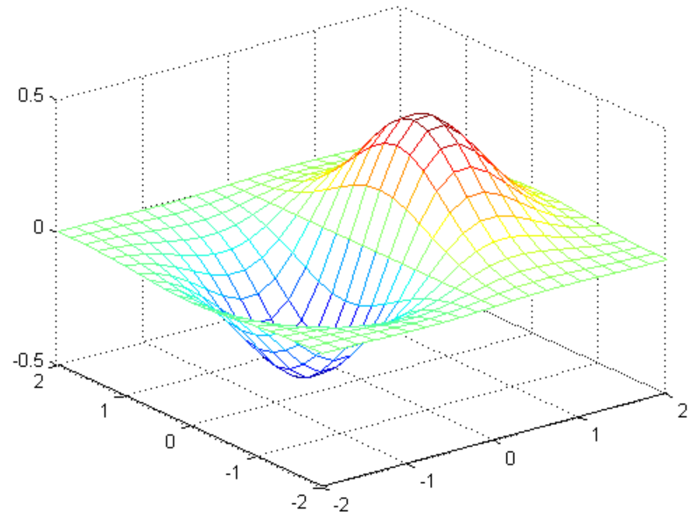
The `ndgrid` function is similar to `meshgrid`, however `ndgrid` supports 1-D to N-D while `meshgrid` is restricted to 2-D and 3-D. The coordinates output by each function are the same, but the shape of the output arrays in the first two dimensions are different. For grid vectors `x1gv`,

x2gv and x3gv of length M, N and P respectively, ndgrid(x1gv, x2gv) will output arrays of size M-by-N while meshgrid(x1gv, x2gv) outputs arrays of size N-by-M. Similarly, ndgrid(x1gv, x2gv, x3gv) will output arrays of size M-by-N-by-P while meshgrid(x1gv, x2gv, x3gv) outputs arrays of size N-by-M-by-P. See "Grid Representation" in the MATLAB Mathematics documentation for more information.

Examples

Evaluate the function $x_1 e^{-x_1^2 - x_2^2}$ over the range $-2 < x_1 < 2, -2 < x_2 < 2$:

```
[X1,X2] = ndgrid(-2:.2:2, -2:.2:2);
Z = X1 .* exp(-X1.^2 - X2.^2);
mesh(X1,X2,Z)
```



See Also

griddedInterpolant | meshgrid | mesh | surf

How To

- "Interpolating Gridded Data"

ndims

Purpose

Number of array dimensions

Syntax

```
n = ndims(A)
```

Description

`n = ndims(A)` returns the number of dimensions in the array `A`. The number of dimensions in an array is always greater than or equal to `2`. Trailing singleton dimensions are ignored. A singleton dimension is any dimension for which `size(A,dim) = 1`.

Algorithms

`ndims(x)` is `length(size(x))`.

See Also

`size`

Purpose

Test for inequality

Syntax

```
A ~= B
ne(A, B)
```

Description

`A ~= B` compares each element of array `A` with the corresponding element of array `B`, and returns an array with elements set to logical 1 (`true`) where `A` and `B` are unequal, or logical 0 (`false`) where they are equal. Each input of the expression can be an array or a scalar value.

If both `A` and `B` are scalar (i.e., 1-by-1 matrices), then the MATLAB software returns a scalar value.

If both `A` and `B` are nonscalar arrays, then these arrays must have the same dimensions, and MATLAB returns an array of the same dimensions as `A` and `B`.

If one input is scalar and the other a nonscalar array, then the scalar input is treated as if it were an array having the same dimensions as the nonscalar input array. In other words, if input `A` is the number 100, and `B` is a 3-by-5 matrix, then `A` is treated as if it were a 3-by-5 matrix of elements, each set to 100. MATLAB returns an array of the same dimensions as the nonscalar input array.

`ne(A, B)` is called for the syntax `A ~= B` when either `A` or `B` is an object.

Examples

Create two 6-by-6 matrices, `A` and `B`, and locate those elements of `A` that are not equal to the corresponding elements of `B`:

```
A = magic(6);
B = repmat(magic(3), 2, 2);

A ~= B
ans =
     1     0     0     1     1     1
     0     1     0     1     1     1
     1     0     0     1     1     1
     0     1     1     1     1     1
     1     0     1     1     1     1
```

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

See Also

`eq` | `le` | `ge` | `lt` | `gt` | relational operators

Purpose

(Will be removed) Point closest to specified location

Note `nearestNeighbor(DelaunayTri)` will be removed in a future release. Use `nearestNeighbor(delaunayTriangulation)` instead.

`DelaunayTri` will be removed in a future release. Use `delaunayTriangulation` instead.

Syntax

```
PI = nearestNeighbor(DT,QX)
PI = nearestNeighbor(DT,QX,QY)
PI = nearestNeighbor(DT,QX,QY,QZ)
[PI,D] = nearestNeighbor(DT,QX,...)
```

Description

`PI = nearestNeighbor(DT,QX)` returns the index of the nearest point in `DT.X` for each query point location in `QX`.

`PI = nearestNeighbor(DT,QX,QY)` and `PI = nearestNeighbor(DT,QX,QY,QZ)` allow the query points to be specified in column vector format when working in 2-D and 3-D.

`[PI,D] = nearestNeighbor(DT,QX,...)` returns the index of the nearest point in `DT.X` for each query point location in `QX`. The corresponding Euclidean distances between the query points and their nearest neighbors are returned in `D`.

Note Note: `nearestNeighbor` is not supported for 2-D triangulations that have constrained edges.

Input Arguments

`DT`	Delaunay triangulation.
`QX`	The matrix `QX` is of size `mpts`-by-`ndim`, `mpts` being the number of query points and `ndim` the dimension of the space where the points reside.

Output Arguments

`PI`	`PI` is a column vector of point indices that index into the points `DT.X`. The length of `PI` is equal to the number of query points `mpts`
`D`	`D` is a column vector of length `mpts`.

Examples

Create a Delaunay triangulation:

```
x = rand(10,1);
y = rand(10,1);
dt = DelaunayTri(x,y);
```

Create query points:

```
qrypts = [0.25 0.25; 0.5 0.5];
```

Find the nearest neighbors to the query points:

```
pid = nearestNeighbor(dt, qrypts)
```

See Also

`pointLocation` | `delaunayTriangulation` | `triangulation`

Purpose

Compare scalar `MException` objects for inequality

Syntax

```
eObj1 ~= eObj2
```

Description

`eObj1 ~= eObj2` tests `MException` objects `eObj1` and `eObj2` for inequality, returning logical 1 (`true`) if the two objects are not identical, otherwise returning logical 0 (`false`).

See Also

`try` | `catch` | `error` | `assert` | `MException` | `isequal(MException)` | `eq(MException)` | `getReport(MException)` | `disp(MException)` | `throw(MException)` | `rethrow(MException)` | `throwAsCaller(MException)` | `addCause(MException)` | `last(MException)`

TriRep.neighbors

Purpose

(Will be removed) Simplex neighbor information

Note `neighbors(TriRep)` will be removed in a future release. Use `neighbors(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
SN = neighbors(TR, SI)
```

Description

`SN = neighbors(TR, SI)` returns the simplex neighbor information for the specified simplices `SI`.

Input Arguments

`TR`	Triangulation representation.
`SI`	`SI` is a column vector of simplex indices that index into the triangulation matrix `TR.Triangulation`. If `SI` is not specified the neighbor information for the entire triangulation is returned, where the neighbors associated with simplex `i` are defined by the `i`'th row of `SN`.

Output Arguments

`SN`	`SN` is an `m`-by-`n` matrix, where `m = length(SI)` , the number of specified simplices, and `n` is the number of neighbors per simplex. Each row `SN(i,:)` represents the neighbors of the simplex `SI(i)`. By convention, the simplex opposite `vertex(j)` of simplex `SI(i)` is `SN(i,j)`. If a simplex has one or more boundary facets, the nonexistent neighbors are represented by `NaN`.

Definitions

A *simplex* is a triangle/tetrahedron or higher-dimensional equivalent. A *facet* is an edge of a triangle or a face of a tetrahedron.

Examples

Example 1

Load a 3-D triangulation and use `TriRep` to compute the neighbors of all tetrahedra.

```
load tetmesh
trep = TriRep(tet, X)
nbrs = neighbors(trep)
```

Example 2

Query a 2-D triangulation created using `DelaunayTri`.

```
x = rand(10,1)
y = rand(10,1)
dt = DelaunayTri(x,y)
```

Find the neighbors of the first triangle:

```
n1 = neighbors(dt, 1)
```

See Also

`delaunayTriangulation | triangulation`

Purpose

Summary of functions in MATLAB .NET interface

Description

Use the following functions to bring assemblies from the Microsoft .NET Framework into the MATLAB environment. The functions are implemented as a package called NET. To use these functions, prefix the function name with package name NET.

BeginInvoke	Initiate asynchronous .NET delegate call
bitnot	Bit-wise NOT
cell	Create cell array
Combine	Convenience function for static .NET System.Delegate Combine method
enableNETfromNetworkDrive	Enable access to .NET commands from network drive
EndInvoke	Retrieve result of asynchronous call initiated by .NET System.Delegate BeginInvoke method
NET.addAssembly	Make .NET assembly visible to MATLAB
NET.Assembly	Members of .NET assembly
NET.convertArray	Convert numeric MATLAB array to .NET array
NET.createArray	Array for nonprimitive .NET types
NET.createGeneric	Create instance of specialized .NET generic type

NET.disableAutoRelease	Lock .NET object representing a RunTime Callable Wrapper (COM Wrapper) so that MATLAB does not release COM object
NET.enableAutoRelease	Unlock .NET object representing a RunTime Callable Wrapper (COM Wrapper) so that MATLAB releases COM object
NET.GenericClass	Represent parameterized generic type definitions
NET.invokeGenericMethod	Invoke generic method of object
NET.isNETSupported	Check for supported Microsoft .NET Framework
NET.NetException	Capture error information for .NET exception
NET.setStaticProperty	Static property or field name
Remove	Convenience function for static .NET System.Delegate Remove method
RemoveAll	Convenience function for static .NET System.Delegate RemoveAll method

How To

- ".NET Libraries"

NET.addAssembly

Purpose

Make .NET assembly visible to MATLAB

Syntax

```
asmInfo = NET.addAssembly(globalName)
asmInfo = NET.addAssembly(privateName)
```

Description

`asmInfo = NET.addAssembly(globalName)` loads a global .NET assembly into MATLAB.

`asmInfo = NET.addAssembly(privateName)` loads a private .NET assembly.

Tips

- MATLAB dynamically loads the `mscorlib.dll` and `system.dll` assemblies from the .NET Framework class library the first time you type "`NET.`" or "`System.`". You do not need to call `NET.addAssembly` to access classes in these assemblies.
- Refer to your .NET product documentation for the name of the assembly and its deployment type (global or private).

Limitations

- `NET.addAssembly` does not support assemblies generated by the MATLAB Builder NE product.

Input Arguments

globalName

One of the following:

- String representing the name of a global assembly.
- Instance of `System.Reflection.AssemblyName` class.

privateName

String representing the full path of a private assembly.

Output Arguments

asmInfo

`NET.Assembly` object containing names of the members of the assembly.

Examples

Display today's date using `System.DateTime` in the `mscorlib` assembly.

```
System.DateTime.Now.ToLongDateString
```

Call the `System.Windows.Forms.MessageBox.Show` method in the global assembly `System.Windows.Forms`.

```
asm = NET.addAssembly('System.Windows.Forms');
import System.Windows.Forms.*;
MessageBox.Show('Simple Message Box')
```

Display classes in the private assembly `NetSample.dll`.

```
asm = NET.addAssembly('c:\work\NetSample.dll');
asm.Classes
```

See Also

`NET.Assembly`

How To

- "An Assembly is a Library of .NET Classes"

Related Links

- MSDN AssemblyName Class

NET.Assembly

Purpose

Members of .NET assembly

Description

`NET.Assembly` object returns names of the members of an assembly.

Construction

The `NET.addAssembly` function creates an instance of this class.

Properties

AssemblyHandle

Instance of `System.Reflection.Assembly` class of the added assembly.

Classes

`nClassx1` cell array of class names of the assembly, where `nClass` is the number of classes

Enums

`nEnumx1` cell array of enums of the assembly, where `nEnum` is the number of enums

Structures

`nStructx1` cell array of structures of the assembly, where `nStruct` is the number of structures

GenericTypes

`nGenTypex1` cell array of generic types of the assembly, where `nGenType` is the number of generic types

Interfaces

`nInterfacex1` cell array of interface names of the assembly, where `nInterface` is the number of interfaces

Delegates

`nDelegatex1` cell array of delegates of the assembly, where `nDelegate` is the number of delegates

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

See Also

`NET.addAssembly`

How To

- "What Classes Are in a .NET Assembly?"

NET.convertArray

Purpose

Convert numeric MATLAB array to .NET array

Note MATLAB automatically converts arrays to .NET types. For information, see "Using Arrays with .NET Applications".

Syntax

`arrObj = NET.convertArray(V,'arrType',[m,n])`

Description

`arrObj = NET.convertArray(V,'arrType',[m,n])` converts a MATLAB array `V` to a .NET array. Optional value `arrType` is a string representing a namespace-qualified .NET array type. Use optional values `m,n` to convert a MATLAB vector to a two-dimensional .NET array (either 1-by-n or m-by-1). If `V` is a MATLAB vector and you do not specify the number of dimensions and their sizes, the output `arrObj` is a one-dimensional .NET array.

If you do not specify `arrType`, MATLAB converts the type according to the "MATLAB Primitive Type Conversion Table".

Examples

Create a list `aList` of random `System.Int32` integers using the `System.Collections.Generic.List` class, and then sort the results:

```
%Create array R of random integers
nInt = 5;
R = randi(100,1,nInt);
%Create .NET array A
A = NET.convertArray(R,'System.Int32');
%Put A into aList, a generic collections list
aList = NET.createGeneric(...
    'System.Collections.Generic.List',...
    {'System.Int32'},A.Length);
aList.AddRange(A);
%Sort the values in aList
aList.Sort;
```

See Also

`NET.createArray`

Purpose

Array for nonprimitive .NET types

Syntax

```
array = NET.createArray(typeName,[m,n,p,...])
array = NET.createArray(typeName,m,n,p,...)
```

Description

`array = NET.createArray(typeName,[m,n,p,...])` creates an `m`-by-`n`-by-`p`-by-`...` `array` of type `typeName`, which is either a fully qualified .NET array type name (`namespace` and array type name) or an instance of the `NET.GenericClass` class, in case of arrays of generic type. `m,n,p,...` are the number of elements in each dimension of the array.

`array = NET.createArray(typeName,m,n,p,...)` alternative syntax for creating an array.

You cannot specify the lower bound of an array.

Examples

Create .NET Array of Generic Type

This example creates a .NET array of `List<Int32>` generic type.

```
genType = NET.GenericClass('System.Collections.Generic.List',...
  'System.Int32');
arr = NET.createArray(genType, 5)

arr = 

  System.Collections.Generic.List<System*Int32>[] handle
  Package: System.Collections.Generic

  Properties:
          Length: 5
      LongLength: 5
            Rank: 1
        SyncRoot: [1x1 System.Collections.Generic.List<System*Int32>
      IsReadOnly: 0
     IsFixedSize: 1
  IsSynchronized: 0
```

```
  Methods, Events, Superclasses
```

Create and Initialize Jagged Array

This example creates a jagged .NET array of 3 elements.

```
jaggedArray = NET.createArray('System.Double[]', 3)

jaggedArray =
  System.Double[][] handle
  Package: System

  Properties:
            Length: 3
        LongLength: 3
              Rank: 1
          SyncRoot: [1x1 System.Double[][]]
        IsReadOnly: 0
       IsFixedSize: 1
    IsSynchronized: 0
  Methods, Events, Superclasses
```

Assign values:

```
jaggedArray(1) = [1, 3, 5, 7, 9];
jaggedArray(2) = [0, 2, 4, 6];
jaggedArray(3) = [11, 22];
```

Access first value of 3rd array:

```
jaggedArray(3,1)

ans =
    11
```

Create Jagged Array of Generic Type

This example creates a jagged array of `List<Double>` generic type.

```
genCls = NET.GenericClass('System.Collections.Generic.List[]',...
  'System.Double');
```

Create the array, `genArr`.

```
genArr = NET.createArray(genCls,3)
```

```
genArr =

  System.Collections.Generic.List<System*Double>[][] handle
  Package: System.Collections.Generic

  Properties:
            Length: 3
        LongLength: 3
              Rank: 1
          SyncRoot: [1x1 System.Collections.Generic.List`1[][][]]
        IsReadOnly: 0
       IsFixedSize: 1
    IsSynchronized: 0

  Methods, Events, Superclasses
```

Create Nested Jagged Array

This command creates a jagged array of type `System.Double[][][]`.

```
netArr = NET.createArray('System.Double[][]', 3)
```

```
netArr =
  System.Double[][][] handle
  Package: System

  Properties:
            Length: 3
        LongLength: 3
              Rank: 1
          SyncRoot: [1x1 System.Double[][][]]
        IsReadOnly: 0
```

```
          IsFixedSize: 1
       IsSynchronized: 0
    Methods, Events, Superclasses
```

See Also

NET.convertArray | NET.createGeneric

Purpose

Create instance of specialized .NET generic type

Syntax

```
genObj = createGeneric(className,paramTypes,
   varargin ctorArgs)
```

Description

`genObj = createGeneric(className,paramTypes,varargin ctorArgs)` creates an instance `genObj` of generic type `className`.

Input Arguments

`className`	Fully qualified string with the generic type name.
`paramTypes`	Allowed cell types are: strings with fully qualified parameter type names and instances of the `NET.GenericClass` class when parameterization with another parameterized type is needed.
`ctorArgs`	Optional, variable length (0 to N) list of constructor arguments matching the arguments of the .NET generic class constructor intended to be invoked.

Output Arguments

`genObj`	Handle to the specialized generic class instance.

Examples

Create a List of System.Double Objects

Create a strongly typed list `dblLst` of objects of type `System.Double`:

```
dblLst =
NET.createGeneric('System.Collections.Generic.List',...
  {'System.Double'},10);
```

Create a List with Key/Value Pairs

Create the `kvpType` generic association where `Key` is of `System.Int32` type and `Value` is a `System.String`:

```
kvpType =
NET.GenericClass('System.Collections.Generic.KeyValuePair',...
  'System.Int32','System.String');
```

Create the list `kvpList` with initial storage capacity for 10 key-value pairs:

```
kvpList =
NET.createGeneric('System.Collections.Generic.List',...
  {kvpType},10);
```

Add an Item to the List

Create a `KeyValuePair` item.

```
kvpItem = NET.createGeneric('System.Collections.Generic.KeyValuePair',...
  {'System.Int32','System.String'},42,'myString');
```

Add this item to the list `kvpList`.

```
kvpList.Add(kvpItem);
```

See Also

`NET.GenericClass`

Purpose

Lock .NET object representing a RunTime Callable Wrapper (COM Wrapper) so that MATLAB does not release COM object

Syntax

`A = NET.disableAutoRelease(obj)`

Description

`A = NET.disableAutoRelease(obj)` locks a .NET object representing a RunTime Callable Wrapper (COM Wrapper) so that MATLAB does not release the COM object. `obj` is a .NET object representing a COM Wrapper.

Before passing a .NET object representing a COM Wrapper to another process, lock the object using this function so that MATLAB does not release it. After using the object, call `NET.enableAutoRelease` to release the COM object.

Examples

The following user-defined function, `GetComApp.m`, has access to a COM object defined in the **pseudo-class**`ComNamespace.ComClass`. One of its methods is `readData`, with the signature:

```
System.String RetVal readData(ComNamespace.ComClass this, System.Strin
```

The input argument is defined in the **pseudo-class** `NetDocTest.MyClass`, which has a property named `MyApp`.

```
function GetComApp(obj)
comObj = ComNamespace.ComClass;
obj.MyApp = comObj;
% To pass a COM object to another process, lock the object
NET.disableAutoRelease(comObj);
end
```

The example in `NET.enableAutoRelease` shows how to call the `GetComApp` function.

See Also

`NET.enableAutoRelease`

How To

- “How MATLAB Handles System.__ComObject”

NET.enableAutoRelease

Purpose

Unlock .NET object representing a RunTime Callable Wrapper (COM Wrapper) so that MATLAB releases COM object

Syntax

`A = NET.enableAutoRelease(obj)`

Description

`A = NET.enableAutoRelease(obj)` releases the COM wrapper when the object goes out of scope, where `obj` is a .NET object representing a COM Wrapper.

Call this function only if the object was locked using `NET.disableAutoRelease`.

Examples

The following **pseudo-code** shows how to call a function (`GetComApp.m`, described in `NET.disableAutoRelease`) which returns a COM object. The object, `mainObj` of type `NetDocTest.MyClass`, has a property, `MyApp`. Call `GetComApp` to get a COM object, and use its `readData` method.

```
mainObj = NetDocTest.MyClass;
GetComApp(mainObj);
app = mainObj.MyApp;
app.readData('hello');
% Unlock the COM object
NET.enableAutoRelease(mainObj.MyApp);
```

See Also

`NET.disableAutoRelease`

How To

- “How MATLAB Handles System.__ComObject”

Purpose

Represent parameterized generic type definitions

Description

Instances of this class are used by the `NET.createGeneric` function when creation of generic specialization requires parameterization with another parameterized type.

Construction

```
genType = NET.GenericClass (className, paramTypes)
```

Input Arguments

className

Fully qualified string containing the generic type name.

paramTypes

Optional, variable length (1 to N) list of types for the generic class parameterization. Allowed argument types are:

- Fully qualified string containing the generic type name.
- Instance of the `NET.GenericClass` class when deeper nested parameterization with another parameterized type is needed.

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create an instance of `System.Collections.Generic.List` of `System.Collections.Generic.KeyValuePair` generic associations where `Key` is of `System.Int32` type and `Value` is a `System.String` class with initial storage capacity for 10 key-value pairs.

```
kvpType = NET.GenericClass('System.Collections.Generic.KeyValuePair',...
  'System.Int32', 'System.String');
kvpList = NET.createGeneric('System.Collections.Generic.List',...
  { kvpType }, 10);
```

See Also

`NET.createGeneric` | `NET.createArray` | `NET.invokeGenericMethod`

How To

- “.NET Generic Classes in MATLAB”

Purpose

Invoke generic method of object

Syntax

```
[varargout] = NET.invokeGenericMethod(obj,
   'genericMethodName', paramTypes, args, ...)
```

Description

`[varargout] = NET.invokeGenericMethod(obj, 'genericMethodName', paramTypes, args, ...)` calls instance or static generic method *`genericMethodName`*.

Input Arguments

`obj`	Allowed argument types are: • Instances of class containing the generic method • Strings with fully qualified class name, if calling static generic methods • Instances of `NET.GenericClass` definitions, if calling static generic methods of a generic class
`genericMethodName`	Generic method name to invoke
`paramTypes`	Cell vector (1 to N) with the types for generic method parameterization, where allowed cell types are: • Strings with fully qualified parameter type name. • Instances of `NET.GenericClass` definitions, if using nested parameterization with another parameterized type
`args`	Optional, variable length (0 to N) list of method arguments

Output Arguments

`varargout` — Variable-length output argument list, `varargout`, from method *`genericMethodName`*

Examples

Use the following syntax to call a generic method that takes two parameterized types and returns a parameterized type:

```
a = NET.invokeGenericMethod(obj, ...
  'myGenericSwapMethod', ...
  {'System.Double', 'System.Double'}, ...
  5, 6);
```

To display generic methods in MATLAB, see the example "Display .NET Generic Methods Using Reflection".

See Also

`NET.GenericClass` | `NET.createGeneric` | `varargout`

How To

- "Call .NET Generic Methods"

Purpose

Check for supported Microsoft .NET Framework

Syntax

```
tf = NET.isNETSupported
```

Description

`tf = NET.isNETSupported` returns logical `1` (`true`) if a supported version of the Microsoft .NET Framework is found. Otherwise, it returns logical `0` (`false`).

NET.NetException

Purpose

Capture error information for .NET exception

Description

Process information from a `NET.NetException` object to handle .NET errors. This class is derived from `MException`.

Construction

`e = NET.NetException(msgID,errMsg,netObj)` constructs instance `e` of `NET.NetException` class.

Input Arguments

msgID

message identifier

errMsg

error message string

netObj

`System.Exception` object that caused the exception

Properties

ExceptionObject

`System.Exception` class causing the error.

Methods

Inherited Methods

See the methods of the base class `MException`.

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

Display error information after trying to load an unknown assembly:

```
try
  NET.addAssembly('C:\Work\invalidfile.dll')
catch e
  e.message;
  if(isa(e, 'NET.NetException'))
    eObj = e.ExceptionObject
```

```
  end
end
```

MATLAB displays:

```
ans =
Message: Could not load file or assembly
  'file:///C:\Work\invalidfile.dll' or
  one of its dependencies. The system cannot
  find the file specified.
Source: mscorlib
HelpLink:

eObj =
  System.IO.FileNotFoundException handle
  Package: System.IO

  Properties:
           Message: [1x1 System.String]
          FileName: [1x1 System.String]
         FusionLog: [1x1 System.String]
              Data: [1x1 System.Collections.ListDictionaryInternal]
    InnerException: []
        TargetSite: [1x1 System.Reflection.RuntimeMethodInfo]
        StackTrace: [1x1 System.String]
          HelpLink: []
            Source: [1x1 System.String]
  Methods, Events, Superclasses
```

See Also

MException

How To

- Class Attributes
- Property Attributes

NET.setStaticProperty

Purpose

Static property or field name

Syntax

```
NET.setStaticProperty('propName', value)
```

Description

`NET.setStaticProperty('propName', value)` sets the static property or field name specified in the string `propName` to the given `value`.

Examples

To set the `myStaticProperty` in the given class and namespace, use the syntax:

```
NET.setStaticProperty('MyTestObject.MyClass.myStaticProperty', 5);
```

Purpose

Create variable in NetCDF file

Syntax

```
nccreate(filename,varname)
nccreate(filename,varname,Name,Value)
```

Description

`nccreate(filename,varname)` creates a scalar double variable named `varname` in the NetCDF file `filename`. If `filename` does not exist, `nccreate` creates the file using the `netcdf4_classic` format. To create a nonscalar variable, use the `Dimensions` argument.

`nccreate(filename,varname,Name,Value)` creates a variable named `varname` with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

filename

Text string specifying a NetCDF file, or the name you want to assign to a new NetCDF file.

varname

Text string specifying the name you want to assign to a variable in a NetCDF file.

Name-Value Pair Arguments

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

'Dimensions'

Cell array specifying dimensions for the variable in the NetCDF file. The cell array lists the dimension name as a string followed by its numerical length: `{dname1,dlength1,dname2,dlength2, ...}`. If a dimension exists, specifying its length is optional. Use `Inf` to specify an unlimited dimension.

All formats other than `netcdf4` format files can have only one unlimited dimension per file and it has to be the last in the list specified. A `netcdf4` format file can have any number of unlimited dimensions in any order.

A single-dimension variable is always treated as a column vector.

'Datatype'

Text string specifying a MATLAB data type. When `nccreate` creates the variable in the NetCDF file, it uses a corresponding NetCDF datatype. The following table describes how MATLAB datatypes map to NetCDF datatypes.

MATLAB Data Type	Corresponding NetCDF Variable Type
double	NC_DOUBLE
single	NC_FLOAT
int64	NC_INT64*
uint64	NC_UINT64*
int32	NC_INT
uint32	NC_UINT*
int16	NC_SHORT
uint16	NC_USHORT*
int8	NC_BYTE
uint8	NC_UBYTE*
char	NC_CHAR

* These data types are only available when the file is a `netcdf4` format file.

'Format'

Text string specifying the type of NetCDF file you want to create.

Format String	Description
classic	NetCDF 3
64bit	NetCDF 3, with 64-bit offsets
netcdf4_classic	NetCDF 4 classic model
netcdf4	NetCDF 4 model (Use this to enable group hierarchy)

If `varname` specifies a group (e.g.,`'/grid3/temperature'`), `nccreate` sets the value of format to `'netcdf4'`.

Default: `netcdf4_classic`

'FillValue'

A scalar specifying the value to use for missing values. To disable fill values, set `FillValue` to the text string `'disable'`. (Available for `netcdf4` or `netcdf4_classic` formats only.)

Default: Value specified by the NetCDF library

'ChunkSize'

Vector specifying the size of the chunk along each dimension, `[num_rows,num_cols,..., num_ndims]`. (Available for `netcdf4` or `netcdf4_classic` formats only.)

Default: Set by the NetCDF library

'DeflateLevel'

Numeric value between 0 (least) and 9 (most) specifying the compression setting for the deflate filter, or the text string `disable`. (Available for `netcdf4` or `netcdf4_classic` formats only.)

Default: Disabled

'Shuffle'

Boolean flag to turn on the shuffle filter. (Available for `netcdf4` or `netcdf4_classic` formats only.)

Default: `False`

Examples

Create a new 2–D variable in a classic format file. Write data to this variable

```
nccreate('myncclassic.nc','peaks',...
         'Dimensions',{'r' 200 'c' 200},...
         'Format','classic');
ncwrite('myncclassic.nc','peaks', peaks(200));
ncdisp('myncclassic.nc');
```

See Also

`ncdisp` | `ncwrite` | `ncinfo` | `ncwriteschema` | `netcdf`

Tutorials

- "Exporting to Network Common Data Form (NetCDF) Files"

Purpose

Display contents of NetCDF data source in Command Window

Syntax

```
ncdisp(source)
ncdisp(source,location)
ncdisp(source,location,modestr)
```

Description

`ncdisp(source)` displays as text in the Command Window, all the groups, dimensions, variable definitions, and all attributes in the NetCDF data source `source`, which can be the name of a NetCDF file or the URL of an OPeNDAP NetCDF data source.

`ncdisp(source,location)` displays information about the variable or group specified by `location` in `source`

`ncdisp(source,location,modestr)` displays the contents of the `location` in `source` according to the value of `modestr`.

Input Arguments

source

Text string specifying the name of the NetCDF file.

location

Text string specifying the location of the variable or group in the NetCDF file. Set `location` to `/` (forward slash) to display the entire contents of the file.

Default: `/`

modestr

Specifies the type of display.

`min`	Display group hierarchy and variable definitions.
`full`	Display group hierarchy with dimensions, attributes, and variable definitions.

Default: `full`

Examples

Visually inspect a NetCDF file.

```
ncdisp('example.nc');
```

Visually inspect a NetCDF file, hiding the attributes.

```
ncdisp('example.nc','/','min');
```

Visually inspect the full details of a variable.

```
ncdisp('example.nc','peaks');
```

See Also

`ncwrite` | `ncinfo` | `ncread` | `netcdf` | `ncreadatt`

Tutorials

- "Importing Network Common Data Form (NetCDF) Files and OPeNDAP Data"

Purpose

Return information about NetCDF data source

Syntax

```
finfo = ncinfo(source)
vinfo = ncinfo(source,varname)
ginfo = ncinfo(source,groupname)
```

Description

`finfo = ncinfo(source)` returns information in the structure `finfo` about the entire NetCDF data source specified by `source`, where source can be the name of a NetCDF file or the URL of an OPeNDAP NetCDF data source.

`vinfo = ncinfo(source,varname)` returns information in the structure `vinfo` about the variable `varname` in `source`.

`ginfo = ncinfo(source,groupname)` returns information in the structure `ginfo` about the group `groupname` in `source` (only NetCDF4 data sources).

Note Use `ncdisp` for visual inspection of a NetCDF source.

Input Arguments

source

Text string specifying the name of a NetCDF file or the URL of an OPeNDAP NetCDF data source.

varname

Text string specifying the name of a variable in a NetCDF file or OPeNDAP data source.

groupname

Text string specifying the name of a group in a NetCDF file or OPeNDAP data source.

Output Arguments

finfo

A structure with the following fields.

Field	Description	
`Filename`	NetCDF file name or OPeNDAP URL	
`Name`	`'/'`, indicating the full file	
`Dimensions`	An array of structures with these fields:	
	`Name`	Dimension name
	`Length`	Current length of dimension
	`Unlimited`	Boolean flag, true for unlimited dimensions
`Variables`	An array of structures with these fields:	
	`Name`	Variable name
	`Dimensions`	Associated dimensions
	`Size`	Current variable size
	`Datatype`	MATLAB datatype
	`Attributes`	Associated variable attributes
	`ChunkSize`	Chunk size, if defined. `[]` otherwise
	`FillValue`	Fill value of the variable.
	`DeflateLevel`	Deflate filter level, if enabled.
	`Shuffle`	Shuffle filter enabled flag
`Attributes`	An array of global attributes with these fields:	
	`Name`	Attribute name
	`Value`	Attribute value

Field	Description	
Groups	An array of groups present in the file, for netcdf4 files; An empty array ([]) for all other NetCDF file formats.	
Format	The format of the NetCDF file	

vinfo

A structure containing only the variable fields from finfo.

Field	Description
Filename	NetCDF file name
Name	Name of the variable
Dimensions	Dimensions of the variable
Size	Size of the current variable
Datatype	MATLAB datatype
Attributes	Attributes associated with the variable
ChunkSize	Chunk size, if defined. [] otherwise.
FillValue	Fill value used in the variable.
DeflateLevel	Deflate filter level, if enabled.
Shuffle	Shuffle filter enabled flag
Format	The format of the NetCDF file

ginfo

A structure containing only the group fields from finfo.

Field	Description
Filename	NetCDF file name
Name	Name of the group

Field	Description
Dimensions	Only dimensions defined in the specified group
Variables	Only variables defined in the specified group
Attributes	Attributes associated with the variable
Groups	Names of groups, if defined. [] otherwise.
Format	The format of the NetCDF file

Examples

Search for dimensions with names that start with the character x in the file.

```
finfo = ncinfo('example.nc');
disp(finfo);
dimNames = {finfo.Dimensions.Name};
dimMatch = strncmpi(dimNames,'x',1);
disp(finfo.Dimensions(dimMatch));
```

Obtain the size of a variable and check if it has any unlimited dimensions.

```
vinfo = ncinfo('example.nc','peaks');
varSize = vinfo.Size;
disp(vinfo);
hasUnLimDim = any([vinfo.Dimensions.Unlimited]);
```

Find all unlimited dimensions defined in a group.

```
ginfo = ncinfo('example.nc','/grid2/');
unlimDims = [ginfo.Dimensions.Unlimited];
disp(ginfo.Dimensions(unlimDims));
```

See Also

`ncdisp` | `ncwrite` | `ncread` | `ncwriteschema` | `netcdf`

Tutorials

- “Importing Network Common Data Form (NetCDF) Files and OPeNDAP Data”

ncread

Purpose

Read data from variable in NetCDF data source

Syntax

```
vardata = ncread(source,varname)
vardata = ncread(source,varname,start,count,stride)
```

Description

`vardata = ncread(source,varname)` reads data from the variable `varname` in `source`, which can be either the name of a NetCDF file or an OPeNDAP NetCDF data source.

`vardata = ncread(source,varname,start,count,stride)` reads data from the variable `varname` in `source` beginning at the location given by `start`. `count` specifies the number of elements to read along the corresponding dimension. The optional argument `stride` specifies the inter-element spacing along each dimension.

Input Arguments

source

Text string specifying the name of a NetCDF file or the URL of an OPeNDAP NetCDF data source.

varname

Text string specifying the name of a variable in the NetCDF file or OPeNDAP NetCDF data source.

start

For an N-dimensional variable, `start` is a vector of length N of indices specifying the starting location. Indices are 1-based.

count

Vector of length N specifying the number of elements to read along the corresponding dimensions. If a particular element of `count` is `Inf`, `ncread` reads data until the end of the corresponding dimension.

stride

Optional argument that specifies the inter-element spacing along each dimension.

Default: Vector of 1s (ones)

Output Arguments

vardata

The data in the variable. `ncread` uses the MATLAB datatype that is the closest type to the corresponding NetCDF datatype, except when at least one of `_FillValue`, `scale_offset`, and `add_offset` variables attribute is present. `ncread` applies the following attribute conventions, in sequence, to `vardata` if the corresponding attribute exists for this variable:

- If the `_FillValue` attribute exists, `ncread` replaces values in `vardata` equal to the value of `_FillValue` with `NaNs`. If the`_FillValue` attribute does not exist, `ncread` queries the NetCDF library for the variable's fill value.
- If the `scale_factor` attribute exists, `ncread` multiplies `vardata` by the value of the `scale_factor` attribute.
- If the `add_offset` attribute exists, `ncread` adds the value of the `add_offset` attribute to `vardata`.

Examples

Read and display the data in the `peaks` variable in the example file.

```
ncdisp('example.nc','peaks');
peaksData  = ncread('example.nc','peaks');
peaksDesc  = ncreadatt('example.nc','peaks','description');
surf(double(peaksData));
title(peaksDesc);
```

Subsample the peaks data by a factor of 2 (read every other value along each dimension).

```
subsetdata = ncread('example.nc','peaks',...
                    [1 1], [Inf Inf], [2 2]);
```

```
surf(double(subsetdata));
```

See Also

ncdisp | ncinfo | netcdf | ncwrite | ncreadatt

Tutorials

- "Importing Network Common Data Form (NetCDF) Files and OPeNDAP Data"

Purpose

Read attribute value from NetCDF data source

Syntax

```
attvalue = ncreadatt(source,location,attname)
```

Description

`attvalue = ncreadatt(source,location,attname)` reads the attribute `attname` from the group or variable specified by `location` in `source`, where source is the name of a NetCDF file or the URL of a NetCDF data source.

Input Arguments

source

Text string specifying the name of a NetCDF file or the URL of an OPeNDAP NetCDF data source.

location

Text string specifying a group or variable in the NetCDF data source. To read global attributes, set `location` to `'/'` (forward slash).

attname

Text string specifying the name of an attribute that you want to read in the NetCDF data source.

Output Arguments

attvalue

Data associated with the attribute.

Examples

Read a global attribute.

```
creation_date = ncreadatt('example.nc','/','creation_date');
disp(creation_date);
```

Read an attribute associated with a variable.

```
scale_factor = ncreadatt('example.nc','temperature','scale_factor');
disp(scale_factor);
```

Read an attribute associated with a group (`netcdf4` format files only).

```
desc_value = ncreadatt('example.nc','/grid2','description');
disp(desc_value);
```

See Also

`ncdisp` | `ncinfo` | `ncread` | `netcdf` | `ncwriteatt`

Tutorials

- “Importing Network Common Data Form (NetCDF) Files and OPeNDAP Data”

Purpose

Write data to NetCDF file

Syntax

```
ncwrite(filename,varname,vardata)
ncwrite(filename,varname,vardata,start,stride)
```

Description

`ncwrite(filename,varname,vardata)` writes the numerical or char data in `vardata` to an existing variable `varname` in the NetCDF file `filename`. `ncwrite` writes the data in `vardata` starting at the beginning of the variable and extends unlimited dimensions automatically, if needed.

If the NetCDF file or the variable do not exist, use `nccreate` to create them first.

`ncwrite(filename,varname,vardata,start,stride)` writes `vardata` to an existing variable `varname` in file `filename` beginning at the location given by `start`. `stride` is an optional argument that specifies the inter-element spacing of the data written. Use this syntax to append data to an existing variable or write partial data.

Tips

- If the variable `varname` already exists, `ncwrite` expects the datatype of `vardata` to match the NetCDF variable data type.
- If the variable `varname` has a `_FillValue`, `scale_factor` or `add_offset` attribute, `ncwrite` expects data in `double` format and casts `vardata` to the NetCDF data type, after applying the following attribute conventions in sequence:

  1. Subtract the value of' the`add_offset` attribute from `vardata`.
  2. Divide `vardata` by the value of the `scale_factor` attribute.
  3. Replace `NaN`s in `vardata` by the value of the `_FillValue` attribute. If this attribute does not exist, `ncwrite` tries to use the fill value for this variable as reported by the NetCDF library.

Input Arguments

filename

Text string specifying the name of a NetCDF file. If the file does not exist, use `nccreate` to create it first.

varname

Text string specifying the name of a variable in a NetCDF file. If the variable does not exist, use `nccreate` to create it first.

vardata

Data to write to the variable in the NetCDF file.

start

For an *N*-dimensional variable, `start` is a vector of indices of length *N* specifying the starting location. Indices are 1-based.

stride

(Optional) Vector of length *N*, specifying the inter-element spacing.

Default: Vector of ones

Examples

Create a new `netcdf4_classic` file, and write a scalar variable with no dimensions. Add the creation time as a global attribute.

```
nccreate('myfile.nc','pi');
ncwrite('myfile.nc','pi',3.1);
ncwriteatt('myfile.nc','/','creation_time',datestr(now));
% overwrite existing data
ncwrite('myfile.nc','pi',3.1416);
ncdisp('myfile.nc');
```

Create a `netcdf4_classic` file with a variable defined on an unlimited dimension. Write data incrementally to the variable.

```
nccreate('myncfile.nc','vmark',...
         'Dimensions', {'time', inf, 'cols', 6},...
         'ChunkSize',  [3 3],...
         'DeflateLevel', 2);
ncwrite('myncfile.nc','vmark', eye(3),[1 1]);
```

```
varData = ncread('myncfile.nc','vmark');
disp(varData);
ncwrite('myncfile.nc','vmark',fliplr(eye(3)),[1 4]);
varData = ncread('myncfile.nc','vmark');
disp(varData);
```

See Also

ncdisp | ncread | ncinfo | netcdf | ncwriteatt | nccreate

Tutorials

- “Exporting to Network Common Data Form (NetCDF) Files”

ncwriteatt

Purpose

Write attribute to NetCDF file

Syntax

`ncwriteatt(filename,location,attname,attvalue)`

Description

`ncwriteatt(filename,location,attname,attvalue)` creates or modifies the attribute specified by `attname` in the group or variable specified by `location`, in the NetCDF file specified by `filename`. `attvalue` can be a numeric vector or a string.

Input Arguments

filename

Text string specifying the name of a NetCDF file

location

Text string specifying a group or variable in the NetCDF file. To write global attributes, set `location` to `'/'` (forward slash).

attname

Text string specifying the name of an existing attribute in a NetCDF file or the name of the attribute that you want to create.

attvalue

Numeric vector or a string.

Examples

Create a global attribute.

```
copyfile(which('example.nc'),'myfile.nc');
fileattrib('myfile.nc','+w');
ncdisp('myfile.nc');
ncwriteatt('myfile.nc','/','creation_date',datestr(now));
ncdisp('myfile.nc');
```

Modify an existing attribute.

```
copyfile(which('example.nc'),'myfile.nc');
fileattrib('myfile.nc','+w');
ncdisp('myfile.nc','peaks');
ncwriteatt('myfile.nc','peaks','description','Output of PEAKS');
ncdisp('myfile.nc','peaks');
```

See Also

`ncdisp` | `ncreadatt` | `ncwrite` | `ncread` | `nccreate` | `netcdf`

Tutorials

- “Exporting to Network Common Data Form (NetCDF) Files”

ncwriteschema

Purpose

Add NetCDF schema definitions to NetCDF file

Syntax

```
ncwriteschema(filename,schema)
```

Description

`ncwriteschema(filename,schema)` creates or adds attributes, dimensions, variable definitions and group structure defined in `schema` to the file `filename`.

Use `ncwriteschema` in combination with `ncinfo` to create a new NetCDF file based on the schema of an existing file. You can also use `ncwriteschema` to add variable definitions, attributes, dimensions, or group structure to an existing file.

Note `ncwriteschema` does not write variable data. Use `ncwrite` to write data to the created variables. Created unlimited dimensions will have an initial size of 0 until you write data.

Note `ncwriteschema` cannot change the format of an existing file. It cannot redefine existing variables and dimensions in `filename`. If your schema contains attributes, dimensions, variable definitions, or a group structure that already exist in the file, `writeschema` issues a warning but continues processing.

Input Arguments

filename

Text string specifying the name of a NetCDF file. If `filename` does not exist, `ncwriteschema` creates a new file using the `netcdf4_classic` format, unless the `Format` field in `schema` specifies another format.

schema

A structure, or array of structures, representing either a dimension, variable, an entire NetCDF file, or a `netcdf4` group. A group or file schema can contain a dimension or variable schema, or both. You can

use the output returned by ncinfo as a schema structure. The following table lists the fields in the various types of schema structures. Optional fields are marked with asterisk (*).

Schema Type	Structure Field	Description
Group/File Schema	Name	Text string identifying the group name. Use '/' to indicate the entire file.
	Dimensions*	Dimension schema
	Variables*	Variable schema
	Attributes*	Structure array of group/global attributes with Name and Value fields
	Format*	Text string identifying a NetCDF file format
Dimension schema	Name	Text string identifying the dimension
	Length	Length of the dimension. Can be Inf.
	Unlimited*	Boolean flag indicating if the dimension is unlimited
	Format*	Text string identifying a NetCDF file format
Variable schema	Name	Text string identifying a variable name
	Dimensions	Variable's dimension schema
	Datatype	Text string identifying a MATLAB datatype
	Attributes*	Structure array of variable attributes with Name and Value fields

Schema Type	Structure Field	Description
	ChunkSize*	Numeric value specifying chunk size of the variable
	FillValue*	Character or numeric fill value
	DeflateValue*	Deflate compression level
	Shuffle*	Boolean flag to turn on the Shuffle filter
	Format*	Text string identifying a NetCDF file format

Examples

Create a classic format file with two dimension definitions.

```
mySchema.Name   = '/';
mySchema.Format = 'classic';
mySchema.Dimensions(1).Name   = 'time';
mySchema.Dimensions(1).Length = Inf;
mySchema.Dimensions(2).Name   = 'rows';
mySchema.Dimensions(2).Length = 10;
ncwriteschema('emptyFile.nc', mySchema);
ncdisp('emptyFile.nc');
```

Create a `netcdf4_classic` format file to store a single variable from an existing file. First use `ncinfo` to get the schema of the peaks variable from the file. Then use `ncwriteschema` to create a NetCDF file, defining the peaks variable. Use `ncread` to get the data associated with the peaks variable and then use `ncwrite` to write the data to the variable in the new NetCDF file.

```
myVarSchema = ncinfo('example.nc','peaks');
ncwriteschema('peaksFile.nc',myVarSchema);
peaksData   = ncread('example.nc','peaks');
ncwrite('peaksFile.nc','peaks',peaksData);
ncdisp('peaksFile.nc');
```

See Also

`ncdisp` | `ncinfo` | `ncwrite` | `ncread` | `netcdf`

Tutorials

- "Exporting to Network Common Data Form (NetCDF) Files"

netcdf

Purpose

Summary of MATLAB Network Common Data Form (NetCDF) capabilities

Description

MATLAB provides both high- and low-level access to NetCDF files:

- High-level access functions make it easy to read a data set from a NetCDF file or write a variable from the MATLAB workspace into a NetCDF file
- Low-level access functions provide interfaces to dozens of functions in the NetCDF library

---

Note For information about MATLAB support for the Common Data Format (CDF), which is a completely separate and incompatible format, see `cdflib`.

High-Level Access

These functions provide high-level access to NetCDF files.

nccreate	Create variable in NetCDF file
ncdisp	Display contents of NetCDF data source in Command Window
ncinfo	Return information about NetCDF data source
ncread	Read data from variable in NetCDF data source
ncreadatt	Read attribute value from NetCDF data source
ncwrite	Write data to NetCDF file
ncwriteatt	Write attribute to NetCDF file
ncwriteschema	Add NetCDF schema definitions to NetCDF file

Low-Level Access

The MATLAB low-level functions provide an API that you can use to enable reading data from and writing data to NetCDF files (known as *data sets* in NetCDF terminology). To use these functions, you should be familiar with the information about NetCDF contained in the `NetCDF C Interface Guide`.

MATLAB supports NetCDF version 4.1.3.

In most cases, the syntax of the MATLAB function matches the syntax of the NetCDF library function. The functions are implemented as a package called `netcdf`. To use these functions, prefix the function name with package name `netcdf`. For example, to call the NetCDF library routine used to open existing NetCDF files, use the following MATLAB syntax:

```
ncid = netcdf.open(ncfile, mode);
```

Library Functions

netcdf	Summary of MATLAB Network Common Data Form (NetCDF) capabilities
netcdf.getChunkCache	Retrieve chunk cache settings for NetCDF library
netcdf.inqLibVers	Return NetCDF library version information
netcdf.setChunkCache	Set default chunk cache settings for NetCDF library
netcdf.setDefaultFormat	Change default netCDF file format

File Operations

netcdf.abort	Revert recent netCDF file definitions
netcdf.close	Close netCDF file

netcdf.create	Create new NetCDF dataset
netcdf.endDef	End netCDF file define mode
netcdf.inq	Return information about netCDF file
netcdf.inqFormat	Determine format of NetCDF file
netcdf.inqGrps	Retrieve array of child group IDs
netcdf.inqUnlimDims	Return list of unlimited dimensions in group
netcdf.open	Open NetCDF data source
netcdf.reDef	Put open netCDF file into define mode
netcdf.setFill	Set netCDF fill mode
netcdf.sync	Synchronize netCDF file to disk

Dimensions

netcdf.defDim	Create netCDF dimension
netcdf.inqDim	Return netCDF dimension name and length
netcdf.inqDimID	Return dimension ID
netcdf.renameDim	Change name of netCDF dimension

Variables

netcdf.defVar	Create NetCDF variable
netcdf.defVarChunking	Define chunking behavior for NetCDF variable
netcdf.defVarDeflate	Define compression parameters for NetCDF variable
netcdf.defVarFill	Define fill parameters for NetCDF variable

netcdf.defVarFletcher32	Define checksum parameters for NetCDF variable
netcdf.getVar	Return data from netCDF variable
netcdf.inqVar	Return information about variable
netcdf.inqVarChunking	Determine chunking settings for NetCDF variable
netcdf.inqVarDeflate	Determine compression settings for NetCDF variable
netcdf.inqVarFill	Determine values of fill parameters for NetCDF variable
netcdf.inqVarFletcher32	Fletcher32 checksum setting for NetCDF variable
netcdf.inqVarID	Return ID associated with variable name
netcdf.putVar	Write data to netCDF variable
netcdf.renameVar	Change name of netCDF variable

Attributes

netcdf.copyAtt	Copy attribute to new location
netcdf.delAtt	Delete netCDF attribute
netcdf.getAtt	Return netCDF attribute
netcdf.inqAtt	Return information about netCDF attribute
netcdf.inqAttID	Return ID of netCDF attribute
netcdf.inqAttName	Return name of netCDF attribute
netcdf.putAtt	Write netCDF attribute
netcdf.renameAtt	Change name of attribute

Utilities

netcdf.getConstant	Return numeric value of named constant
netcdf.getConstantNames	Return list of constants known to netCDF library

Purpose Revert recent netCDF file definitions

Syntax

```
netcdf.abort(ncid)
```

Description

`netcdf.abort(ncid)` reverts a netCDF file to its previous state, backing out any definitions made since the file last entered define mode. A file enters define mode when you create it (using `netcdf.create`) or when you explicitly enter define mode (using `netcdf.redef`). Once you leave define mode (using `netcdf.endDef`), you cannot revert the definitions you made while in define mode. `ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`. A call to `netcdf.abort` closes the file.

This function corresponds to the `nc_abort` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example creates a new file, performs an operation on the file, and then reverts the file back to its original state. To run this example, you must have write permission in your current directory.

```
% Create a netCDF file
ncid = netcdf.create('foo.nc','NC_NOCLOBBER');

% Perform an operation, such as defining a dimension.
dimid = netcdf.defDim(ncid, 'lat', 50);

% Revert the file back to its previous state.
netcdf.abort(ncid)

% Verify that the file is now closed.
dimid = netcdf.defDim(ncid, 'lat', 50); % should fail
??? Error using ==> netcdflib
NetCDF: Not a valid ID

Error in ==> defDim at 22
dimid = netcdflib('def_dim', ncid,dimname,dimlen);
```

See Also

`netcdf.create` | `netcdf.endDef` | `netcdf.reDef`

Purpose

Close netCDF file

Syntax

```
netcdf.close(ncid)
```

Description

`netcdf.close(ncid)` terminates access to the netCDF file identified by `ncid`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_close` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example creates a new netCDF file, and then closes the file. You must have write permission in your current directory to run this example.

```
ncid = netcdf.open('foo.nc','NC_WRITE')

netcdf.close(ncid)
```

See Also

`netcdf.create` | `netCDF.open`

netcdf.copyAtt

Purpose

Copy attribute to new location

Syntax

```
netcdf.copyAtt(ncid_in,varid_in,attname,ncid_out,varid_out)
```

Description

`netcdf.copyAtt(ncid_in,varid_in,attname,ncid_out,varid_out)` copies an attribute from one variable to another, possibly across files. `ncid_in` and `ncid_out` are netCDF file identifiers returned by `netcdf.create` or `netcdf.open`. `varid_in` identifies the variable with an attribute that you want to copy. `varid_out` identifies the variable to which you want to associate a copy of the attribute.

This function corresponds to the `nc_copy_att` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example makes a copy of the attribute associated with the first variable in the netCDF example file, `example.nc`, in a new file. To run this example, you must have write permission in your current directory.

```
% Open example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get identifier for a variable in the file.
varid = netcdf.inqVarID(ncid,'avagadros_number');

% Create new netCDF file.
ncid2 = netcdf.create('foo.nc','NC_NOCLOBBER');

% Define a dimension in the new file.
dimid2 = netcdf.defDim(ncid2,'x',50);

% Define a variable in the new file.
varid2 = netcdf.defVar(ncid2,'myvar','double',dimid2);

% Copy the attribute named 'description' into the new file,
% associating the attribute with the new variable.
netcdf.copyAtt(ncid,varid,'description',ncid2,varid2);
```

```
%
% Check the name of the attribute in new file.
attname = netcdf.inqAttName(ncid2,varid2,0)

attname =

description
```

See Also

netcdf.inqAtt | netcdf.inqAttID | netcdf.inqAttName | netcdf.putAtt | netcdf.renameAtt

netcdf.create

Purpose

Create new NetCDF dataset

Syntax

```
ncid = netcdf.create(filename, mode)
[chunksize_out, ncid]=netcdf.create(filename,mode,initsz,
   chunksize)
```

Description

`ncid = netcdf.create(filename, mode)` creates a new NetCDF file according to the file creation mode. The return value `ncid` is a file ID. The `mode` parameter is a text string that describes the type of file access, which can have any of the following values.

Value	Description
`NOCLOBBER`	Prevent overwriting of existing file with the
`CLOBBER`	Overwrite any existing file with the same name.
`SHARE`	Allow synchronous file updates.
`64BIT_OFFSET`	Allow easier creation of files and variables which are larger than two gigabytes.
`NETCDF4`	Create a NetCDF-4/HDF5 file
`CLASSIC_MODEL`	Enforce the classic model; has no effect unless used in a bitwise-or with `NETCDF4`

Note You can specify the mode as a numeric value, retrieved using the `netcdf.getConstant` function. To specify more than one mode, use a bitwise-OR of the numeric values of the modes.

`[chunksize_out, ncid]=netcdf.create(filename,mode,initsz,chunksize)` creates a new netCDF file, but with additional performance tuning parameters. `initsz` sets the initial size of the file. `chunksize` can affect I/O performance. The actual value chosen by the NetCDF library might not correspond to the input value.

This function corresponds to the `nc_create` and `nc__create` functions in the NetCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example creates a NetCDF dataset named `foo.nc`, only if no other file with the same name exists in the current directory. To run this example, you must have write permission in your current directory.

```
ncid = netcdf.create('foo.nc','NOCLOBBER');
```

See Also

`netcdf.getConstant` | `netcdf.open`

netcdf.defDim

Purpose

Create netCDF dimension

Syntax

```
dimid = netcdf.defDim(ncid,dimname,dimlen)
```

Description

`dimid = netcdf.defDim(ncid,dimname,dimlen)` creates a new dimension in the netCDF file specified by `ncid`, where`dimname` is a character string that specifies the name of the dimension and `dimlen` is a numeric value that specifies its length. To define an unlimited dimension, specify the predefined constant `'NC_UNLIMITED'` for `dimlen`, using `netcdf.getConstant` to retrieve the value.

`netcdf.defDim` returns `dimid`, a numeric ID corresponding to the new dimension.

This function corresponds to the `nc_def_dim` function in the netCDF library C API.To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

Examples

Create a new file and define two dimensions in the file. One dimension is an unlimited dimension. To run this example, you must have write permission in your current folder.

```
% Create a netCDF file.
ncid = netcdf.create('foo.nc','NC_NOCLOBBER')

% Define a dimension.
lat_dimID = netcdf.defDim(ncid,'latitude',360);

% Define an unlimited dimension.
long_dimID = netcdf.defDim(ncid,'longitude',netcdf.getConstant('NC_UNLIMI
```

See Also

`netcdf.getConstant`

netcdf.defGrp

Purpose

Create group in NetCDF file

Syntax

```
childGrpID = netcdf.defGrp(parentGroupId,childGroupName)
```

Description

`childGrpID = netcdf.defGrp(parentGroupId,childGroupName)` creates a child group with the name specified by `childGroupName`, that is the child of the parent group specified by `parentGroupId`

Input Arguments

parentGroupId

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

childGroupName

Text string specifying the name that you want to assign to the group.

Output Arguments

childGrpID

Identifier of a NetCDF group.

Examples

This example creates a NetCDF dataset and then defines a group.

```
ncid = netcdf.create('myfile.nc','netcdf4');
childGroupId = netcdf.defGrp(ncid,'mygroup');
netcdf.close(ncid);
```

References

This function corresponds to the `nc_def_grp` function in the NetCDF library C API.

For copyright information, read the files `netcdfcopyright.txt` and `mexnccopyright.txt`.

See Also

`netcdf` | `netcdf.inqGrps`

netcdf.defVar

Purpose

Create NetCDF variable

Syntax

`varid = netcdf.defVar(ncid,varname,xtype,dimids)`

Description

`varid = netcdf.defVar(ncid,varname,xtype,dimids)` creates a new variable in the dataset identified by `ncid`.

`varname` is a character string that specifies the name of the variable. `xtype` can be either a character string specifying the data type of the variable, such as `'double'`, or it can be the numeric equivalent returned by the `netcdf.getConstant` function. `dimids` specifies a list of dimension IDs.

`netcdf.defVar` returns `varid`, a numeric identifier for the new variable.

This function corresponds to the `nc_def_var` function in the NetCDF library C API. Because MATLAB uses FORTRAN-style ordering, the fastest-varying dimension comes first and the slowest comes last. Any unlimited dimension is therefore last in the list of dimension IDs. This ordering is the reverse of that found in the C API.To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example creates a new netCDF file, defines a dimension in the file, and then defines a variable on that dimension. (In netCDF files, you must create a dimension before you can create a variable.) To run this example, you must have write permission in your current directory.

```
% Create  netCDF file.
ncid = netcdf.create('foo.nc','NC_NOCLOBBER');
%
% Define a dimension in the new file.
dimid = netcdf.defDim(ncid,'x',50);

% Define a variable in the new file.
varid = netcdf.defVar(ncid,'myvar','double',dimid)
```

See Also

`netCDF.getConstant` | `netCDF.inqVar` | `netCDF.putVar`

Purpose

Define chunking behavior for NetCDF variable

Syntax

```
netcdf.defVarChunking(ncid,varid,storage,chunkDims)
```

Description

`netcdf.defVarChunking(ncid,varid,storage,chunkDims)`. sets the chunk settings for the variable specified by `varid`. Chunking is a technique to improve performance. `storage` specifies the type of chunking to use and `chunkDims` specifies the extents of the chunk size. You must specify the chunk size used with a variable after creating the variable but before you write data to the variable.

You cannot specify the chunk size for variables in a NetCDF file created with the netCDF-3 mode (`CLASSIC_MODEL`).

Input Arguments

ncid

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

varid

Identifier of a NetCDF variable, returned by `netcdf.defVar`.

storage

Text string specifying whether NetCDF should break the variable into chunks when writing to a file. If set to `CHUNKED`, NetCDF breaks the variable into chunks; if set to `CONTIGUOUS`, NetCDF does not break the data into chunks.

chunkDims

Array specifying the dimensions of the chunk.

Because MATLAB uses FORTRAN-style ordering, the order of dimensions in `chunkdims` is reversed relative to what would be in the C API.

If storage is `CONTIGUOUS`, you can omit `chunkDims`.

Default: Chunk size determined by the NetCDF library.

Examples

This example creates a NetCDF file and specifies the chunking behavior of a variable.

```
ncid = netcdf.create('myfile.nc','NETCDF4');
latdimid = netcdf.defDim(ncid,'lat',1800);
londimid = netcdf.defDim(ncid,'col',3600);
varid = netcdf.defVar(ncid,'earthgrid','double',[latdimid londimid]);
netcdf.defVarChunking(ncid,varid,'CHUNKED',[180 360]);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_def_var_chunking` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.inqVarChunking`

Purpose

Define compression parameters for NetCDF variable

Syntax

`netcdf.defVarDeflate(ncid,varid,shuffle,deflate,deflateLevel)`

Description

`netcdf.defVarDeflate(ncid,varid,shuffle,deflate,deflateLevel)` sets the compression parameters for the NetCDF variable specified by `varid` in the location specified by `ncid`.

Input Arguments

ncid

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

varid

Identifier of a NetCDF variable, returned by `netcdf.defVar`.

shuffle

Boolean value. To turn on the shuffle filter, set this argument to `true`. The shuffle filter can assist with the compression of integer data by changing the byte order in the data stream.

deflate

Boolean value. To turn on compression, set this argument to `true` and set the `deflateLevel` argument to the desired compression level.

deflateLevel

Numeric value between `0` and `9` specifying the amount of compression, where `0` is no compression and `9` is the most compression.

Examples

This example create a variable with dimensions [1800 3600] and a compression level of 5. This results in a chunked layout that is a 10-by-10 grid.

```
ncid = netcdf.create('myfile.nc','NETCDF4');
latdimid = netcdf.defDim(ncid,'lat',1800);
```

```
londimid = netcdf.defDim(ncid,'col',3600);
varid = netcdf.defVar(ncid,'earthgrid','double',[latdimid londimid]);
netcdf.defVarDeflate(ncid,varid,true,true,5);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_def_var_deflate` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.inqVarDeflate`

Purpose

Define fill parameters for NetCDF variable

Syntax

`netcdf.defVarFill(ncid,varid,noFillMode,fillValue)`

Description

`netcdf.defVarFill(ncid,varid,noFillMode,fillValue)` sets the fill parameters for the NetCDF variable identified by `varid`. `ncid` specifies the location.

For netCDF-4 files, you can only specify fill values when the NetCDF is in definition mode (before calling `netcdf.endDef`). For NetCDF files in classic and 64-bit offset modes, you can turn no-fill mode on and off at any time.

Input Arguments

ncid

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

varid

Identifier of a NetCDF variable, returned by `netcdf.defVar`.

noFillMode

Boolean value. When set to `true`, turns off use of fill values for the variable, which can be helpful in high performance applications. When `true`, `netcdf.defVarFill` ignores the value of the `fillValue` argument. To use the fill value, set this to `false`.

fillValue

Specifies the value to use in the variable when no other value is specified. The data type must be the same data type as the variable.

Examples

This example creates a NetCDF file and defines a fill value for a variable.

```
ncid = netcdf.create('myfile.nc','NETCDF4');
dimid = netcdf.defDim(ncid,'latitude',180);
```

```
varid = netcdf.defVar(ncid,'latitude','double',dimid);
netcdf.defVarFill(ncid,varid,false,-999);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_def_var_fill` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.setFill` | `netcdf.inqVarFill`

Purpose

Define checksum parameters for NetCDF variable

Syntax

`netcdf.defVarFletcher32(ncid,varid,setting)`

Description

`netcdf.defVarFletcher32(ncid,varid,setting)` defines the checksum settings for the NetCDF variable specified by `varid` in the file specified by `ncid`.

Input Arguments

ncid

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

varid

Identifier of a NetCDF variable, returned by `netcdf.defVar`.

setting

Text string specifying whether Fletcher32 checksum error detection is used with the variable. To turn on Fletcher32 checksum, specify the value `FLETCHER32`. To turn off the use of checksum error detection, specify the value `NOCHECKSUM`.

Examples

This example creates a NetCDF dataset and turns on the Fletcher32 checksum for a variable.

```
ncid = netcdf.create('myfile.nc','NETCDF4');
latdimid = netcdf.defDim(ncid,'lat',1800);
londimid = netcdf.defDim(ncid,'col',3600);
varid = netcdf.defVar(ncid,'earthgrid','double',[latdimid londimid]);
netcdf.defVarFletcher32(ncid,varid,'FLETCHER32');
netcdf.close(ncid);
```

References

This function corresponds to the `nc_def_var_fletcher32` function in the NetCDF library C API.

For copyright information, read the files `netcdfcopyright.txt` and `mexnccopyright.txt`.

See Also

`netcdf` | `netcdf.inqVarFletcher32`

netcdf.delAtt

Purpose

Delete netCDF attribute

Syntax

`netcdf.delAtt(ncid,varid,attName)`

Description

`netcdf.delAtt(ncid,varid,attName)` deletes the attribute identified by the text string `attName`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`varid` is a numeric value that identifies the variable. To delete a global attribute, use `netcdf.getConstant('GLOBAL')` for the `varid`. You must be in define mode to delete an attribute.

This function corresponds to the `nc_del_att` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open a netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Determine number of global attributes in file.
[numdims numvars numatts unlimdimID] = netcdf.inq(ncid);

numatts =

   1

% Get name of attribute; it is needed for deletion.
attname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0)

% Put file in define mode to delete an attribute.
netcdf.reDef(ncid);

% Delete the global attribute in the netCDF file.
```

```
netcdf.delAtt(ncid,netcdf.getConstant('GLOBAL'),attname);

% Verify that the global attribute was deleted.
[numdims numvars numatts unlimdimID] = netcdf.inq(ncid);

numatts =

   0
```

See Also

netcdf.getConstant | netcdf.inqAttName

Purpose

End netCDF file define mode

Syntax

```
netcdf.endDef(ncid)
netcdf.endDef(ncid,h_minfree,v_align,v_minfree,r_align)
```

Description

`netcdf.endDef(ncid)` takes a netCDF file out of define mode and into data mode. `ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`netcdf.endDef(ncid,h_minfree,v_align,v_minfree,r_align)` takes a netCDF file out of define mode, specifying four additional performance tuning parameters. For example, one reason for using the performance parameters is to reserve extra space in the netCDF file header using the `h_minfree` parameter:

```
ncid = netcdf.endDef(ncid,20000,4,0,4);
```

This reserves 20,000 bytes in the header, which can be used later when adding attributes. This can be extremely efficient when working with very large files. To understand how to use these performance tuning parameters, see the netCDF library documentation.

This function corresponds to the `nc_enddef` and `nc__enddef` functions in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

When you create a file using `netcdf.create`, the functions opens the file in define mode. This example uses `netcdf.endDef` to take the file out of define mode.

```
% Create a netCDF file.
ncid = netcdf.create('foo.c','NC_NOCLOBBER');

% Define a dimension.
dimid = netcdf.defDim(ncid, 'lat', 50);

% Leave define mode.
netcdf.endDef(ncid)
```

```
% Test if still in define mode.
dimid = netcdf.defDim(ncid, 'lon', 50);  % should fail
??? Error using ==> netcdflib
NetCDF: Operation not allowed in data mode

Error in ==> defDim at 22
dimid = netcdflib('def_dim', ncid,dimname,dimlen);
```

See Also

netcdf.create | netcdf.reDef

Purpose

Return netCDF attribute

Syntax

```
attrvalue = netcdf.getAtt(ncid,varid,attname)
attrvalue = netcdf.getAtt(ncid,varid,attname,output_datatype)
```

Description

`attrvalue = netcdf.getAtt(ncid,varid,attname)` returns `attrvalue`, the value of the attribute specified by the text string `attname`. When it chooses the data type of `attrvalue`, MATLAB attempts to match the netCDF class of the attribute. For example, if the attribute has the netCDF data type `NC_INT`, MATLAB uses the `int32` class for the output data. If an attribute has the netCDF data type `NC_BYTE`, the class of the output data is `int8` value.

`attrvalue = netcdf.getAtt(ncid,varid,attname,output_datatype)` returns `attrvalue`, the value of the attribute specified by the text string `attname`, using the output class specified by `output_datatype`. You can specify any of the following strings for the output data type.

'int'	'double'	'int16'
'short'	'single'	'int8'
'float'	'int32'	'uint8'

This function corresponds to several attribute I/O functions in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`, and gets the value of the attribute associated with the first variable. The example also gets the value of the global variable in the file.

```
% Open a netCDF file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get name of first variable.
```

```
[varname vartype vardimIDs varatts] = netcdf.inqVar(ncid,0);

% Get ID of variable, given its name.
varid = netcdf.inqVarID(ncid,varname);

% Get attribute name, given variable id.
attname = netcdf.inqAttName(ncid,varid,0);

% Get value of attribute.
attval = netcdf.getAtt(ncid,varid,attname);

% Get name of global attribute
gattname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0);

% Get value of global attribute.
gattval = netcdf.getAtt(ncid,netcdf.getConstant('NC_GLOBAL'),gattname)

gattval =

09-Jun-2008
```

See Also

netcdf.inqAtt | netcdf.putAtt

netcdf.getChunkCache

Purpose

Retrieve chunk cache settings for NetCDF library

Syntax

```
[csize, nelems, premp] = netcdf.getChunkCache()
```

Description

`[csize, nelems, premp] = netcdf.getChunkCache()` returns the default chunk cache settings.

Output Arguments

csize

Scalar double specifying the total size of the raw data chunk cache in bytes.

nelems

Scalar double specifying the number of chunk slots in the raw data chunk cache hash table.

premp

Double, between 0 and 1, inclusive, that specifies how the library handles preempting fully read chunks in the chunk cache. A value of zero means fully read chunks are treated no differently than other chunks, that is, preemption occurs solely based on the Least Recently Used (LRU) algorithm. A value of 1 means fully read chunks are always preempted before other chunks.

Examples

Determine information about the chunk cache size used by the NetCDF library.

```
[csize, nelems, premp] = netcdf.getChunkCache();
```

References

This function corresponds to the `nc_get_chunk_cache` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.setChunkCache`

netcdf.getConstant

Purpose

Return numeric value of named constant

Syntax

```
val = netcdf.getConstant(param_name)
```

Description

`val = netcdf.getConstant(param_name)` returns the numeric value corresponding to the name of a constant defined by the netCDF library. For example, `netcdf.getConstant('NC_NOCLOBBER')` returns the numeric value corresponding to the netCDF constant `NC_NOCLOBBER`.

The value for `param_name` can be either upper- or lowercase, and does not need to include the leading three characters `'NC_'`. To retrieve a list of all the names defined by the netCDF library, use the `netcdf.getConstantNames` function.

This function has no direct equivalent in the netCDF C interface. To find out more about NetCDF, see `netcdf`.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`.

```
% Open example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Determine contents of the file.
[ndims nvars natts dimm] = netcdf.inq(ncid);

% Get name of global attribute.
% Note: You must use netcdf.getConstant to specify NC_GLOBAL.
attname = netcdf.inqattname(ncid,netcdf.getConstant('NC_GLOBAL'),0)

attname =

creation_date
```

See Also

`netcdf.getConstantNames`

Purpose

Return list of constants known to netCDF library

Syntax

`val = netcdf.getConstantNames(param_name)`

Description

`val = netcdf.getConstantNames(param_name)` returns a list of names of netCDF library constants, definitions, and enumerations. When these strings are supplied as actual parameters to MATLAB netCDF package functions, the functions automatically convert the constant to the appropriate numeric value.

This MATLAB function has no direct equivalent in the netCDF C interface. To find out more about netCDF, see `netcdf`.

Examples

```
nc_constants = netcdf.getConstantNames

nc_constants =

    'NC2_ERR'
    'NC_64BIT_OFFSET'
    'NC_BYTE'
    'NC_CHAR'
    'NC_CLOBBER'
    'NC_DOUBLE'
    'NC_EBADDIM'
    'NC_EBADID'
    'NC_EBADNAME'
    'NC_EBADTYPE'
  ...
```

See Also

`netCDF.getConstantNames`

netcdf.getVar

Purpose

Return data from netCDF variable

Syntax

```
data = netcdf.getVar(ncid,varid)
data = netcdf.getVar(ncid,varid,start)
data = netcdf.getVar(ncid,varid,start,count)
data = netcdf.getVar(ncid,varid,start,count,stride)
data = netcdf.getVar(...,output_type)
```

Description

`data = netcdf.getVar(ncid,varid)` returns `data`, the value of the variable specified by `varid`. MATLAB attempts to match the class of the output data to netCDF class of the variable.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`data = netcdf.getVar(ncid,varid,start)` returns a single value starting at the specified index, `start`.

`data = netcdf.getVar(ncid,varid,start,count)` returns a contiguous section of a variable. `start` specifies the starting point and `count` specifies the amount of data to return.

`data = netcdf.getVar(ncid,varid,start,count,stride)` returns a subset of a section of a variable. `start` specifies the starting point, `count` specifies the extent of the section, and `stride` specifies which values to return.

`data = netcdf.getVar(...,output_type)` specifies the data type of the return value `data`. You can specify any of the following strings for the output data type.

'int8'
'uint8'
'int16'
'int32'
'single'
'double'

This function corresponds to several functions in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

Open the example netCDF file and get the value of a variable in the file.

```
% Open example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get the name of the first variable.
[varname, xtype, varDimIDs, varAtts] = netcdf.inqVar(ncid,0);

% Get variable ID of the first variable, given its name.
varid = netcdf.inqVarID(ncid,varname);

% Get the value of the first variable, given its ID.
data = netcdf.getVar(ncid,varid)

data =

  6.0221e+023

% Determine the data type of the output value
whos data
 Name      Size            Bytes  Class     Attributes

  data      1x1                 8  double

% Get the value again, this time specifying the output data type
data = netcdf.getVar(ncid,varid,'single');

% Determine the data type of the output value
whos data
  Name      Size            Bytes  Class     Attributes

  data      1x1                 4  single
```

See Also

`netcdf.create` | `netcdf.inqVarID` | `netcdf.open`

Purpose

Return information about netCDF file

Syntax

```
[ndims,nvars,ngatts,unlimdimid] = netcdf.inq(ncid)
```

Description

`[ndims,nvars,ngatts,unlimdimid] = netcdf.inq(ncid)` returns the number of dimensions, variables, and global attributes in a netCDF file. The function also returns the ID of the dimension defined with unlimited length, if one exists.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`. You can call `netcdf.inq` in either define mode or data mode.

This function corresponds to the `nc_inq` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`, and uses the `netcdf.inq` function to get information about the contents of the file.

```
% Open netCDF example file.
ncid = netcdf.open('example.nc','NC_NOWRITE')

% Get information about the contents of the file.
[numdims, numvars, numglobalatts, unlimdimID] = netcdf.inq(ncid)

numdims =

     4


numvars =

     4


numglobalatts =
```

```
     1


unlimdimID =

     3
```

See Also

netcdf.create | netcdf.open

Purpose

Retrieve list of dimension identifiers in group

Syntax

```
dimIDs = netcdf.inqDimIDs(ncid)
dimIDs = netcdf.inqDimIDs(ncid,includeParents)
```

Description

`dimIDs = netcdf.inqDimIDs(ncid)` returns a list of dimension identifiers in the group specified by `ncid`.

`dimIDs = netcdf.inqDimIDs(ncid,includeParents)` includes all dimensions in all parent groups if `includeParents` is `true`.

Input Arguments

ncid

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

includeParents

Boolean value. If set to `true`, `netcdf.inqDimIDs` includes the dimensions of all parent groups.

Default: `false`

Output Arguments

dimIDs

Array of dimension IDs

Examples

This example opens the NetCDF sample file and gets the IDs of all the dimensions.

```
ncid = netcdf.open('example.nc','NOWRITE');
gid = netcdf.inqNcid(ncid,'grid1');
dimids = netcdf.inqDimIDs(gid);
dimids_all = netcdf.inqDimIDS(gid, true);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_dimids` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.inqVarIDs`

Purpose

Determine format of NetCDF file

Syntax

```
format = netcdf.inqFormat(ncid)
```

Description

`format = netcdf.inqFormat(ncid)` returns the format for the file specified by NetCDF file identifier, `ncid`.

Input Arguments

ncid

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Output Arguments

format

Text string that specifies the format of the NetCDF file. Values include:

Format String	Description
`FORMAT_CLASSIC`	Classic format — Original NetCDF format, used by all NetCDF files created between 1989 and 2004
`FORMAT_64BIT`	Classic format, 64–bit — Original format with 64–bit addressing capability to allow creation and access of much larger files.
`FORMAT_NETCDF4`	Enhanced model, HDF5-based — Introduced in 2008, NetCDF, version 4, extends the classic model and is based on HDF5.
`FORMAT_NETCDF4_CLASSIC`	Classic model, HDF5-based — Introduced in 2008, NetCDF, version 4, implements classic model but is based on HDF5.

Examples

This example opens the sample NetCDF file and determines the format.

```
ncid = netcdf.open('example.nc','NOWRITE');
fmt = netcdf.inqFormat(ncid)
```

```
format =

FORMAT_NETCDF4

netcdf.close(ncid);
```

References

This function corresponds to the nc_inq_format function in the NetCDF library C API.

For copyright information, read the netcdfcopyright.txt and mexnccopyright.txt files.

See Also

netcdf.getConstant | netcdf

Purpose

Retrieve name of group

Syntax

```
groupName = netcdf.inqGrpName(ncid)
```

Description

`groupName = netcdf.inqGrpName(ncid)` returns the name of a group specified by `ncid`.

Input Arguments

ncid

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Output Arguments

groupName

Text string specifying name of a group. The root group has the name `'/'`.

Examples

This example opens the NetCDF sample file and gets the names of groups in the dataset.

```
ncid = netcdf.open('example.nc','nowrite');
name = netcdf.inqGrpName(ncid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_grpname` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.inqGrpNameFull`

netcdf.inqGrpNameFull

Purpose

Complete pathname of group

Syntax

`groupName = netcdf.inqGrpNameFull(ncid)`

Description

`groupName = netcdf.inqGrpNameFull(ncid)` returns the complete pathname of the group specified by `ncid`.

Input Arguments

ncid

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Output Arguments

groupName

Text string specifying complete path of group.

The root group has the name `'/'`. The names of parent groups and child groups use the forward slash `'/'` separator, as in UNIX folder names, for example, `/group1/subgrp2/subsubgrp3`.

Examples

Open the NetCDF sample dataset and retrieve the names of all groups.

```
ncid = netcdf.open('example.nc','NOWRITE');
gid = netcdf.inqNcid(ncid,'grid2');
fullName = netcdf.inqGrpNameFull(gid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_grpname_full` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.inqGrpName`

Purpose

Retrieve ID of parent group.

Syntax

`parentGroupID = netcdf.inqGrpParent(ncid)`

Description

`parentGroupID = netcdf.inqGrpParent(ncid)` returns the ID of the parent group given the location of the child group, specified by `ncid`.

Input Arguments

ncid

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Output Arguments

parentGroupID

Identifier of the NetCDF group or file that is the parent of the specified file or group.

Examples

This example opens the NetCDF sample file and gets the full path of the parent of the specified group.

```
ncid = netcdf.open('example.nc','NOWRITE');
gid = netcdf.inqNcid(ncid,'grid2');
parentId = netcdf.inqGrpParent(gid);
fullName = netcdf.inqGrpNameFull(parentId);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_grp_parent` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.inqGrps`

netcdf.inqGrps

Purpose

Retrieve array of child group IDs

Syntax

```
childGrps = netcdf.inqGrps(ncid)
```

Description

`childGrps = netcdf.inqGrps(ncid)` returns all the child group IDs in the parent group, specified by `ncid`.

Input Arguments

ncid

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Output Arguments

childGrps

Array containing identifiers of child groups in the specified NetCDF file or group.

Examples

This example opens the sample NetCDF file and then gets information about the groups it contains.

```
ncid = netcdf.open('example.nc','nowrite');
childGroups = netcdf.inqGrps(ncid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_grps` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.inqNcid`

Purpose

Return ID of named group

Syntax

```
childGroupId = netcdf.inqNcid(ncid,childGroupName)
```

Description

`childGroupId = netcdf.inqNcid(ncid,childGroupName)` returns the ID of the child group, specified by the name `childGroupName`, in the file or group specified by `ncid`.

Input Arguments

ncid

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

childGroupName

Text string specifying the name of a NetCDF group.

Output Arguments

childGroupID

Identifier of a NetCDF group.

Examples

This example opens the sample NetCDF dataset and then gets the ID of a group in the dataset.

```
ncid = netcdf.open('example.nc','nowrite');
gid = netcdf.inqNcid(ncid,'grid1');
netcdf.close(ncid);
```

References

This function corresponds to the nc_inq_ncid function in the netCDF library C API. Read the files `netcdfcopyright.txt` and `mexnccopyright.txt` for more information.

See Also

`netcdf` | `netcdf.inqGrpName` | `netcdf.inqGrpNameFull`

netcdf.inqUnlimDims

Purpose

Return list of unlimited dimensions in group

Syntax

```
unlimdimIDs = netcdf.inqUnlimDims(ncid)
```

Description

`unlimdimIDs = netcdf.inqUnlimDims(ncid)` returns the IDs of all unlimited dimensions in the group specified by `ncid`.

Input Arguments

ncid

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or group, returned by `netcdf.defGrp`.

Output Arguments

unlimDimIDs

An array containing the identifiers of each unlimited dimension. `unlimDimIDs` is empty if there are no unlimited dimensions.

Examples

This example opens the NetCDF sample dataset and gets the IDs of all the unlimited dimensions.

```
ncid = netcdf.open('example.nc','NOWRITE');
dimids = netcdf.inqUnlimDims(ncid)

dimids =

     []

netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_unlim_dims` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.defDim` | `netcdf.inqDim` | `netcdf.inqDimID` | `netcdf.renameDim` | `netcdf.inqDimIDs`

netcdf.inqVarIDs

Purpose

IDs of all variables in group

Syntax

```
varids = netcdf.inqVarIDs(ncid)
```

Description

`varids = netcdf.inqVarIDs(ncid)` returns IDs of the all the variables in the group specified by `ncid`.

Input Arguments

ncid

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

Output Arguments

varids

Array containing identifiers of variables in a NetCDF file or group.

Examples

This example opens the NetCDF sample file and gets the IDs of all the variables in a group.

```
ncid = netcdf.open('example.nc','NOWRITE');
gid = netcdf.inqNcid(ncid,'grid1');
varids = netcdf.inqVarIDs(gid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_varids` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.inqDimIDs` | `netcdf.inqVarID`

netcdf.inqVarChunking

Purpose

Determine chunking settings for NetCDF variable

Syntax

```
[storage,chunkSizes] = netcdf.inqVarChunking(ncid,varid)
```

Description

`[storage,chunkSizes] = netcdf.inqVarChunking(ncid,varid)` returns the type of chunking and the dimensions of a chunk for the NetCDF variable specified by `varid`, in the file or group specified by `ncid`.

Input Arguments

ncid

Identifier of NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

varid

Identifier of NetCDF variable, returned by `netcdf.defVar`.

Output Arguments

storage

Text string specifying if NetCDF breaks the data into chunks when writing to a file. `CHUNKED` indicates the data is chunked; `CONTIGUOUS` indicates that the data is not chunked.

chunkSizes

Array specifying the dimensions of the chunk.

Because MATLAB uses FORTRAN-style ordering, the order of dimensions in `chunkdims` is reversed relative to what would be in the NetCDF C API.

If the storage type specified is `CONTIGUOUS`, `netcdf.inqVarChunking` returns an empty array, `[]`.

Examples

This example opens the NetCDF sample dataset and gets the values of chunking parameters associated with a variable.

```
ncid = netcdf.open('example.nc','NOWRITE');
```

```
groupid = netcdf.inqNcid(ncid,'grid1');
varid = netcdf.inqVarID(groupid,'temp');
[storage,chunkSize] = netcdf.inqVarChunking(groupid,varid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_var_chunking` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.defVar` | `netcdf.defVarChunking`

netcdf.inqVarDeflate

Purpose

Determine compression settings for NetCDF variable

Syntax

```
[shuffle,deflate,deflateLevel] = netcdf.inqVarDeflate(ncid,
   varid)
```

Description

`[shuffle,deflate,deflateLevel] = netcdf.inqVarDeflate(ncid,varid)` returns the compression parameters for the NetCDF variable specified by `varid` in the location specified by `ncid`.

Input Arguments

ncid

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or of a NetCDF group, returned by `netcdf.defGrp`.

varid

Identifier of NetCDF variable, returned by `netcdf.defVar`.

Output Arguments

shuffle

Boolean value. `true` indicates that the shuffle filter is enabled for the specified variable. The shuffle filter can assist with the compression of integer data by changing the byte order in the data stream.

deflate

Boolean value. `true` indicates that compression is enabled for this variable. The `deflateLevel` argument specifies the level of compression.

deflateLevel

Scalar value between `0` and `9` specifying the amount of compression, where `0` is no compression and `9` is the most compression

Examples

This example opens the NetCDF sample file and gets information about variable compression.

```
ncid = netcdf.open('example.nc','NOWRITE');
groupid = netcdf.inqNcid(ncid,'grid1');
varid = netcdf.inqVarID(groupid,'temp');
[shuffle,deflate,deflateLevel] = netcdf.inqVarDeflate(groupid,varid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_var_deflate` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.defVarDeflate`

netcdf.inqVarFill

Purpose

Determine values of fill parameters for NetCDF variable

Syntax

```
[noFillMode,fillValue] = netcdf.inqVarFill(ncid,varid)
```

Description

`[noFillMode,fillValue] = netcdf.inqVarFill(ncid,varid)` returns the fill mode and the fill value for the variable `varid` in the file or group specified by `ncid`.

Input Arguments

ncid

Identifier of a NetCDF file, returned by `netcdf.create` or `netcdf.open`, or a NetCDF group, returned by `netcdf.defGrp`.

varid

Identifier of NetCDF variable.

Output Arguments

noFillMode

Boolean value. `true` indicates that use of the fill values for the variable has been disabled.

fillValue

Specifies the value to use in the variable when no other value is specified and use of fill values has been enabled.

Examples

This example opens the NetCDF sample dataset and gets the fill mode and fill value used with a variable.

```
ncid = netcdf.open('example.nc','NOWRITE');
varid = netcdf.inqVarID(ncid,'temperature');
[noFillMode,fillValue] = netcdf.inqVarFill(ncid,varid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_var_fill` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.defVarFill` | `netcdf.setFill`

netcdf.inqVarFletcher32

Purpose

Fletcher32 checksum setting for NetCDF variable

Syntax

```
setting = netcdf.inqVarFletcher32(ncid,varid)
```

Description

`setting = netcdf.inqVarFletcher32(ncid,varid)` returns the Fletcher32 checksum setting for the NetCDF variable specified by `varid` in the file or group specified by `ncid`.

Input Arguments

ncid

Identifier for NetCDF file, returned by `netcdf.create` or `netcdf.open`, or group, returned by `netcdf.defGrp`.

varid

Identifier of NetCDF variable.

Output Arguments

setting

Text string specifying whether the Fletcher32 checksum is turned on for the specified variable. `netcdf.inqVarFletcher32` returns the text string `FLETCHER32` if the checksum is turned on for the variable; otherwise, `NOCHECKSUM`.

Examples

This example opens the sample NetCDF file and gets information about the checksum setting for a variable.

```
ncid = netcdf.open('example.nc','NOWRITE');
varid = netcdf.inqVarID(ncid,'temperature');
setting = netcdf.inqVarFletcher32(ncid,varid);
netcdf.close(ncid);
```

References

This function corresponds to the `nc_inq_var_fletcher32` function in the netCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` for more information.

See Also

`netcdf` | `netcdf.defVarFletcher32`

netcdf.inqAtt

Purpose

Return information about netCDF attribute

Syntax

```
[xtype,attlen] = netcdf.inqAtt(ncid,varid,attname)
```

Description

`[xtype,attlen] = netcdf.inqAtt(ncid,varid,attname)` returns the data type, `xtype`, and length, `attlen`, of the attribute identified by the text string `attname`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`varid` identifies the variable that the attribute is associated with. To get information about a global attribute, specify `netcdf.getConstant('NC_GLOBAL')` in place of `varid`.

This function corresponds to the `nc_inq_att` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`, and gets information about an attribute in the file.

```
% Open netCDF example file.
ncid = netcdf.open('example.nc','NOWRITE');

% Get identifier of a variable in the file, given its name.
varid = netcdf.inqVarID(ncid,'avagadros_number');

% Get attribute name, given variable id and attribute number.
attname = netcdf.inqAttName(ncid,varid,0);

% Get information about the attribute.
[xtype,attlen] = netcdf.inqAtt(ncid,varid,'description')

xtype =

     2
```

```
attlen =

    31

% Get name of global attribute
gattname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0);

% Get information about global attribute.
[gxtype gattlen] = netcdf.inqAtt(ncid,netcdf.getConstant('NC_GLOBAL'),

gxtype =

     2


gattlen =

    11
```

See Also

netcdf.inqAttID | netcdf.inqAttName

netcdf.inqAttID

Purpose

Return ID of netCDF attribute

Syntax

```
attnum = netcdf.inqAttID(ncid,varid,attname)
```

Description

`attnum = netcdf.inqAttID(ncid,varid,attname)` retrieves `attnum`, the identifier of the attribute specified by the text string `attname`.

`varid` specifies the variable the attribute is associated with.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_attid` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens the netCDF example file included with MATLAB, `example.nc`.

```
% Open the netCDF example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get the identifier of a variable in the file.
varid = netcdf.inqVarID(ncid,'avagadros_number');

% Retrieve the identifier of the attribute associated with the variable.
attid = netcdf.inqAttID(ncid,varid,'description');
```

See Also

`netcdf.inqAtt` | `netcdf.inqAttName`

Purpose

Return name of netCDF attribute

Syntax

```
attname = netcdf.inqAttName(ncid,varid,attnum)
```

Description

`attname = netcdf.inqAttName(ncid,varid,attnum)` returns `attname`, a text string specifying the name of an attribute.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`varid` is a numeric identifier of a variable in the file. If you want to get the name of a global attribute in the file, use `netcdf.getConstant('NC_GLOBAL')` in place of `attnum` is a zero-based numeric value specifying the attribute, with 0 indicating the first attribute, 1 the second attribute, and so on.

This function corresponds to the `nc_inq_attname` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get identifier of a variable in the file.
varid = netcdf.inqVarID(ncid,'avagadros_number')

% Get the name of the attribute associated with the variable.
attname = netcdf.inqAttName(ncid,varid,0)

attname =

description

% Get the name of the global attribute associated with the variable.
gattname = netcdf.inqAttName(ncid,netcdf.getConstant('NC_GLOBAL'),0)
```

```
gattname =

creation_date
```

See Also

netcdf.inqAtt | netcdf.inqAttID

Purpose

Return netCDF dimension name and length

Syntax

```
[dimname, dimlen] = netcdf.inqDim(ncid,dimid)
```

Description

`[dimname, dimlen] = netcdf.inqDim(ncid,dimid)` returns the name, `dimname`, and length, `dimlen`, of the dimension specified by `dimid`. If `ndims` is the number of dimensions defined for a netCDF file, each dimension has an ID between 0 and `ndims-1`. For example, the dimension identifier of the first dimension is 0, the second dimension is 1, and so on.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_dim` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

The example opens the example netCDF file include with MATLAB, `example.nc`.

```
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get name and length of first dimension
[dimname, dimlen] = netcdf.inqDim(ncid,0)

dimname =

x


dimlen =

    50
```

See Also

`netcdf.inqDimID`

netcdf.inqDimID

Purpose

Return dimension ID

Syntax

```
dimid = netcdf.inqDimID(ncid,dimname)
```

Description

`dimid = netcdf.inqDimID(ncid,dimname)` returns `dimid`, the identifier of the dimension specified by the character string `dimname`. You can use the `netcdf.inqDim` function to retrieve the dimension name. `ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_dimid` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF example file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get name and length of first dimension
[dimname, dimlen] = netcdf.inqDim(ncid,0);

% Retrieve identifier of dimension.
dimid = netcdf.inqDimID(ncid,dimname)

dimid =

     0
```

See Also

`netcdf.inqDim`

netcdf.inqLibVers

Purpose

Return NetCDF library version information

Syntax

```
libvers = netcdf.inqLibVers
```

Description

`libvers = netcdf.inqLibVers` returns a string identifying the version of the NetCDF library.

This function corresponds to the `nc_inq_libvers` function in the NetCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

```
libvers = netcdf.inqLibVers

libvers =

4.1.3
```

netcdf.inqVar

Purpose Return information about variable

Syntax

```
[varname,xtype,dimids,natts] = netcdf.inqVar(ncid,varid)
```

Description

`[varname,xtype,dimids,natts] = netcdf.inqVar(ncid,varid)` returns the name, data type, dimensions IDs, and the number of attributes associated with the variable identified by `varid`. Dimension IDs are zero-based.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_var` function in the netCDF library C API. Because MATLAB uses FORTRAN-style ordering, however, the order of the dimension IDs is reversed relative to what would be obtained from the C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`, and gets information about a variable in the file.

```
% Open the example netCDF file.
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get information about third variable in the file.
[varname, xtype, dimids, numatts] = netcdf.inqVar(ncid,2)

varname =

peaks


xtype =

     5
```

```
dimids =

     0     1


numatts =

     1     1
```

See Also

netcdf.create | netcdf.inqVarID | netcdf.open

netcdf.inqVarID

Purpose

Return ID associated with variable name

Syntax

```
varid = netcdf.inqVarID(ncid,varname)
```

Description

`varid = netcdf.inqVarID(ncid,varname)` returns `varid`, the ID of a netCDF variable specified by the text string, `varname`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

This function corresponds to the `nc_inq_varid` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens the example netCDF file included with MATLAB, `example.nc`, and uses several inquiry functions to get the ID of the first variable.

```
ncid = netcdf.open('example.nc','NC_NOWRITE');

% Get information about first variable in the file.
[varname, xtype, dimids, atts] = netcdf.inqVar(ncid,0);

% Get variable ID of the first variable, given its name
varid = netcdf.inqVarID(ncid,varname)

varid =

     0
```

See Also

`netcdf.create` | `netcdf.inqVar` | `netcdf.open`

Purpose

Open NetCDF data source

Syntax

```
ncid = netcdf.open(source)
ncid = netcdf.open(source,mode)
[chosen_chunksize, ncid] = netcdf.open(source,mode,chunksize)
```

Description

`ncid = netcdf.open(source)` opens `source`, which can be the name of a NetCDF file or the URL of an OPeNDAP NetCDF data source, for read-only access. Returns a NetCDF ID in `ncid`.

`ncid = netcdf.open(source,mode)` opens `source` with the type of access specified by `mode`, which can have any of the following values.

Value	Description
'WRITE'	Read-write access
'SHARE'	Synchronous file updates
'NOWRITE'	Read-only access (Default)

You can also specify `mode` as a numeric value that can be retrieved using `netcdf.getConstant`. Use these numeric values when you want to specify a bitwise-OR of several modes.

`[chosen_chunksize, ncid] = netcdf.open(source,mode,chunksize)` opens `source`, an existing netCDF data source, specifying the additional I/O performance tuning parameter, `chunksize`. The actual value used by the NetCDF library might not correspond to the input value you specify.

This function corresponds to the `nc_open` and `nc__open` functions in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens the example NetCDF file included with MATLAB, `example.nc`.

```
ncid = netcdf.open('example.nc','NOWRITE');
netcdf.close(ncid);
```

See Also

`netcdf.close` | `netcdf` | `netcdf.getConstant`

Purpose

Write netCDF attribute

Syntax

```
netcdf.putAtt(ncid,varid,attrname,attrvalue)
```

Description

`netcdf.putAtt(ncid,varid,attrname,attrvalue)` writes the attribute named `attrname` with value `attrvalue` to the netCDF variable specified by `varid`. To specify a global attribute, use `netcdf.getConstant('NC_GLOBAL')` for `varid`

`ncid` is a netCDF file identifier returned by `netCDF.create` or `netCDF.open`.

This function corresponds to several attribute I/O functions in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example creates a new netCDF file, defines a dimension and a variable, adds data to the variable, and then creates an attribute associated with the variable. To run this example, you must have writer permission in your current directory.

```
% Create a variable in the workspace.
my_vardata = linspace(0,50,50);

% Create a netCDF file.
ncid = netcdf.create('foo.nc','NC_WRITE');

% Define a dimension in the file.
dimid = netcdf.defDim(ncid,'my_dim',50);

% Define a new variable in the file.
varid = netcdf.defVar(ncid,'my_var','double',dimid);

% Leave define mode and enter data mode to write data.
netcdf.endDef(ncid);

% Write data to variable.
netcdf.putVar(ncid,varid,my_vardata);
```

```
% Re-enter define mode.
netcdf.reDef(ncid);

% Create an attribute associated with the variable.
netcdf.putAtt(ncid,0,'my_att',10);

% Verify that the attribute was created.
[xtype xlen] = netcdf.inqAtt(ncid,0,'my_att')

xtype =

     6


xlen =

     1
```

See Also

netcdf.getatt

Purpose

Write data to netCDF variable

Syntax

```
netcdf.putVar(ncid,varid,data)
netcdf.putVar(ncid,varid,start,data)
netcdf.putVar(ncid,varid,start,count,data)
netcdf.putVar(ncid,varid,start,count,stride,data)
```

Description

`netcdf.putVar(ncid,varid,data)` writes `data` to a netCDF variable identified by `varid`.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`netcdf.putVar(ncid,varid,start,data)` writes a single data value into the variable at the index specified by `start`.

`netcdf.putVar(ncid,varid,start,count,data)` writes a section of values into the netCDF variable at the index specified by the vector `start` to the extent specified by the vector `count`, along each dimension of the specified variable.

`netcdf.putVar(ncid,varid,start,count,stride,data)` writes the subsection specified by sampling interval,`stride`, of the values in the section of the variable beginning at the index `start` and to the extent specified by `count`.

This function corresponds to several variable I/O functions in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

Write Variable to New netCDF File

Create a new netCDF file and write a variable to the file.

Create a 50 element vector for a variable.

```
my_vardata = linspace(0,50,50);
```

Open the netCDF file.

```
ncid = netcdf.create('foo.nc','NC_WRITE');
```

Define the dimensions of the variable.

```
dimid = netcdf.defDim(ncid,'my_dim',50);
```

Define a new variable in the file.

```
my_varID = netcdf.defVar(ncid,'my_var','double',dimid);
```

Leave define mode and enter data mode to write data.

```
netcdf.endDef(ncid);
```

Write data to variable.

```
netcdf.putVar(ncid,my_varID,my_vardata);
```

Verify that the variable was created.

```
[varname xtype dimid natts ] = netcdf.inqVar(ncid,0)
```

```
varname =

my_var


xtype =

     6


dimid =

     0


natts =
```

```
0
```

Close the file.

```
netcdf.close(ncid)
```

See Also

netcdf.getVar

netcdf.reDef

Purpose

Put open netCDF file into define mode

Syntax

`netcdf.reDef(ncid)`

Description

`netcdf.reDef(ncid)` puts an open netCDF file into define mode so that dimensions, variables, and attributes can be added or renamed. Attributes can also be deleted in define mode. `ncid` is a valid NetCDF file ID, returned from a previous call to `netcdf.open` or `netcdf.create`.

This function corresponds to the `nc_redef` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example opens a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open a netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Try to define a dimension.
dimid = netcdf.defdim(ncid, 'lat', 50); % should fail.
??? Error using ==> netcdflib
NetCDF: Operation not allowed in data mode

Error in ==> defDim at 22
dimid = netcdflib('def_dim', ncid,dimname,dimlen);

% Put file in define mode.
netcdf.reDef(ncid);

% Try to define a dimension again. Should succeed.
dimid = netcdf.defDim(ncid, 'lat', 50);
```

See Also

`netcdf.create` | `netcdf.endDef` | `netcdf.open`

Purpose

Change name of attribute

Syntax

`netcdf.renameAtt(ncid,varid,oldName,newName)`

Description

`netcdf.renameAtt(ncid,varid,oldName,newName)` changes the name of the attribute specified by the character string `oldName`.

`newName` is a character string that specifies the new name.

`ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`.

`varid` identifies the variable to which the attribute is associated. To specify a global attribute, use `netcdf.getConstant('NC_GLOBAL')` for `varid`.

This function corresponds to the `nc_rename_att` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example modifies a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Get the ID of a variable the attribute is associated with.
varID = netcdf.inqVarID(ncid,'avagadros_number')

% Rename the attribute.
netcdf.renameAtt(ncid,varID,'description','Description');

% Verify that the name changed.
attname = netcdf.inqAttName(ncid,varID,0)

attname =

Description
```

See Also

`netcdf.inqAttName`

netcdf.renameDim

Purpose

Change name of netCDF dimension

Syntax

```
netcdf.renameDim(ncid,dimid,newName)
```

Description

`netcdf.renameDim(ncid,dimid,newName)` renames the dimension identified by the dimension identifier, `dimid`.

`newName` is a character string specifying the new name. `ncid` is a netCDF file identifier returned by `netcdf.create` or `netcdf.open`

This function corresponds to the `nc_rename_dim` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This examples modifies a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Put file is define mode.
netcdf.reDef(ncid)

% Get the identifer of a dimension to rename.
dimid = netcdf.inqDimID(ncid,'x');

% Rename the dimension.
netcdf.renameDim(ncid,dimid,'Xdim')

% Verify that the name changed.
data = netcdf.inqDim(ncid,dimid)

data =

Xdim
```

See Also

`netcdf.defDim`

netcdf.renameVar

Purpose Change name of netCDF variable

Syntax `netcdf.renameVar(ncid,varid,newName)`

Description `netcdf.renameVar(ncid,varid,newName)` renames the variable identified by `varid` in the netCDF file identified by `ncid`.`newName` is a character string specifying the new name.

This function corresponds to the `nc_rename_var` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples This example modifies a local copy of the example netCDF file included with MATLAB, `example.nc`.

```
% Open netCDF file.
ncid = netcdf.open('my_example.nc','NC_WRITE')

% Put file in define mode.
netcdf.redef(ncid)

% Get name of first variable
[varname, xtype, varDimIDs, varAtts] = netcdf.inqVar(ncid,0);

varname

varname =

avagadros_number

% Rename the variable, using a capital letter to start the name.
netcdf.renameVar(ncid,0,'Avagadros_number')

% Verify that the name of the variable changed.
[varname, xtype, varDimIDs, varAtts] = netcdf.inqVar(ncid,0);

varname
```

```
varname =

Avagadros_number
```

See Also

netCDF.defVar | netCDF.inqVar | netCDF.putVar

netcdf.setChunkCache

Purpose

Set default chunk cache settings for NetCDF library

Syntax

```
netcdf.setChunkCache(csize,nelems,premp)
```

Description

`netcdf.setChunkCache(csize,nelems,premp)` sets the default chunk cache settings used by the NetCDF library.

Settings apply for subsequent file open or create operations, for the remainder of the MATLAB session or until you issue a `clear mex` call. This function does not change the chunk cache settings of files already open.

Input Arguments

csize

Scalar double specifying the total size of the raw data chunk cache in bytes.

nelems

Scalar double specifying the number of chunk slots in the raw data chunk cache hash table.

premp

Scalar double, between `0` and `1`, inclusive, that specifies how the library handles preempting fully read chunks in the chunk cache. A value of `0` means fully read chunks are treated no differently than other chunks, that is, preemption occurs solely based on the Least Recently Used (LRU) algorithm. A value of 1 means fully read chunks are always preempted before other chunks.

Examples

This example sets the cache chunk size used by the NetCDF library.

```
netcdf.setChunkCache(32000000, 2003, .75)
```

References

This function corresponds to the `nc_set_chunk_cache` function in the NetCDF library C API.

For copyright information, read the `netcdfcopyright.txt` and `mexnccopyright.txt` files.

See Also

`netcdf` | `netcdf.getChunkCache`

netcdf.setDefaultFormat

Purpose

Change default netCDF file format

Syntax

```
oldFormat = netcdf.setDefaultFormat(newFormat)
```

Description

`oldFormat = netcdf.setDefaultFormat(newFormat)` changes the default format used by `netCDF.create` when creating new netCDF files, and returns the value of the old format. You can use this function to change the format used by a netCDF file without having to change the creation mode flag used in each call to `netCDF.create`.

`newFormat` can be either of the following values.

Value	Description
'NC_FORMAT_CLASSIC'	Original netCDF file format
'NC_FORMAT_64BIT'	64-bit offset format; relaxes limitations on creating very large files

You can also specify the numeric equivalent of these values, as retrieved by `netcdf.getConstant`.

This function corresponds to the `nc_set_default_format` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

```
oldFormat = netcdf.setDefaultFormat('NC_FORMAT_64BIT');
```

See Also

`netcdf.create`

Purpose

Set netCDF fill mode

Syntax

```
old_mode = netcdf.setFill(ncid,new_mode)
```

Description

`old_mode = netcdf.setFill(ncid,new_mode)` sets the fill mode for a netCDF file identified by `ncid`.

`new_mode` can be either `'FILL'` or `'NOFILL'` or their numeric equivalents, as retrieved by `netcdf.getConstant`. The default mode is `'FILL'`. netCDF pre-fills data with fill values. Specifying `'NOFILL'` can be used to enhance performance, because it avoids the duplicate writes that occur when the netCDF writes fill values that are later overwritten with data.

This function corresponds to the `nc_set_fill` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example creates a new file and specifies the fill mode used by netCDF with the file.

```
ncid = netcdf.open('foo.nc','NC_WRITE');

% Set filling behavior
old_mode = netcdf.setFill(ncid,'NC_NOFILL');
```

See Also

`netcdf.getConstant`

netcdf.sync

Purpose

Synchronize netCDF file to disk

Syntax

```
netcdf.sync(ncid)
```

Description

`netcdf.sync(ncid)` synchronizes the state of a netCDF file to disk. The netCDF library normally buffers accesses to the underlying netCDF file, unless you specify the `NC_SHARE` mode when you opened the file with `netcdf.open` or `netcdf.create`. To call `netcdf.sync`, the netCDF file must be in data mode.

This function corresponds to the `nc_sync` function in the netCDF library C API. To use this function, you should be familiar with the netCDF programming paradigm. See `netcdf` for more information.

Examples

This example creates a new netCDF file for write access, performs an operation on the file, takes the file out of define mode, and then synchronizes the file to disk.

```
% Create a netCDF file.
ncid = netcdf.create('foo.nc','NC_WRITE');

% Perform an operation.
dimid = netcdf.defDim(ncid,'Xdim',50);

% Take file out of define mode.
netcdf.endDef(ncid);

% Synchronize the file to disk.
netcdf.sync(ncid)
```

See Also

`netcdf.close` | `netcdf.create` | `netcdf.open` | `netcdf.endDef`

Purpose

Determine where to draw graphics objects

Syntax

```
newplot
h = newplot
h = newplot(hsave)
```

Description

`newplot` prepares a figure and axes for subsequent graphics commands.

`h = newplot` prepares a figure and axes for subsequent graphics commands and returns a handle to the current axes.

`h = newplot(hsave)` prepares and returns an axes, but does not delete any objects whose handles you have assigned to the `hsave` argument, which can be a vector of handles. If `hsave` is not empty, the figure and axes containing `hsave` are prepared for plotting instead of the current axes of the current figure. If `hsave` is empty, `newplot` behaves as if it were called without any inputs.

Tips

Use `newplot` at the beginning of high-level graphics code to determine which figure and axes to target for graphics output. Calling `newplot` can change the current figure and current axes. Basically, there are three options when you are drawing graphics in existing figures and axes:

- Add the new graphics without changing any properties or deleting any objects.
- Delete all existing objects whose handles are not hidden before drawing the new objects.
- Delete all existing objects regardless of whether or not their handles are hidden, and reset most properties to their defaults before drawing the new objects (refer to the following table for specific information).

The figure and axes `NextPlot` properties determine how `newplot` behaves. The following two tables describe this behavior with various property values.

First, `newplot` reads the current figure's `NextPlot` property and acts accordingly.

NextPlot	What Happens
`new`	Create a new figure and use it as the current figure.
`add`	Draw to the current figure without clearing any graphics objects already present.
`replacechildren`	Remove all child objects whose `HandleVisibility` property is set to `on` and reset figure `NextPlot` property to `add`. This clears the current figure and is equivalent to issuing the `clf` command.
`replace`	Remove all child objects (regardless of the setting of the `HandleVisibility` property) and reset figure properties to their defaults, except `NextPlot` is reset to `add` regardless of user-defined defaults. • `Position`, `Units`, `PaperPosition`, and `PaperUnits` are not reset. This clears and resets the current figure and is equivalent to issuing the `clf reset` command.

After `newplot` establishes which figure to draw in, it reads the current axes' `NextPlot` property and acts accordingly.

NextPlot	Description
`add`	Draw into the current axes, retaining all graphics objects already present.
`replacechildren`	Remove all child objects whose `HandleVisibility` property is set to `on`, but do not reset axes properties. This clears the current axes like the `cla` command.
`replace`	Remove all child objects (regardless of the setting of the `HandleVisibility` property) and reset axes properties to their defaults, except `Position` and `Units`. This clears and resets the current axes like the `cla reset` command.

See Also

`axes` | `cla` | `clf` | `figure` | `hold` | `ishold` | `reset` | figure | axes

How To

- Controlling Graphics Output

Tiff.nextDirectory

Purpose

Make next IFD current IFD

Syntax

```
tiffobj.nextDirectory()
```

Description

`tiffobj.nextDirectory()` makes the next image file directory (IFD) in the file the current IFD. `Tiff` object methods operate on the current IFD. Use this method to navigate among IFDs in a TIFF file containing multiple images.

Examples

Open a `Tiff` object and change the current IFD to the next IFD in the file. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path. The TIFF file should contain multiple images.

```
t = Tiff('myfile.tif', 'r');
t.nextDirectory();
```

References

This method corresponds to the `TIFFReadDirectory` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff.setDirectory`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

Purpose

Exponent of next higher power of 2

Syntax

```
P = nextpow2(A)
```

Description

`P = nextpow2(A)` returns the exponents for the smallest powers of two that satisfy

$$2^p \geq |A|$$

for each element in `A`.

You can use `nextpow2` to pad the signal you pass to `fft`. Doing so can speed up the computation of the FFT when the signal length is not an exact power of 2.

Input Arguments

A - Input values

scalar, vector, or array of real numbers

Input values, specified as a scalar, vector, or array of real numbers of any numeric type.

Example: `15`

Example: `[-15.123 32.456 63.111]`

Example: `int16([-15 32 63])`

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Examples

Next Power of 2 of Double Integer Values

Define a vector of `double` integer values and calculate the exponents for the next power of 2 higher than those values.

```
a = [1 -2 3 -4 5 9 519];
p = nextpow2(a)

p =
```

```
     0     1     2     2     3     4    10
```

Calculate the positive next powers of 2.

```
np2 = 2.^p

np2 =

     1     2     4     4     8    16  1024
```

Preserve the sign of the original input values.

```
np2.*sign(a)

ans =

      1    -2     4    -4     8    16  1024
```

Next Power of 2 of Unsigned Integer Values

Define a vector of unsigned integers and calculate the exponents for the next power of 2 higher than those values.

```
a = uint32([1020 4000 32700]);
p = nextpow2(a)

p =

          10          12          15
```

Calculate the next powers of 2 higher than the values in a.

```
2.^p

ans =

        1024        4096       32768
```

Optimize FFT with Padding

Use `nextpow2` to increase the performance of `fft` when the length of your signal is not a power of 2.

Create a 1-D vector containing 8191 sample values.

```
x = gallery('uniformdata',[1,8191],0);
```

Calculate the next power of 2 higher than 8191.

```
p = nextpow2(8191);
n = 2^p
```

```
n =

        8192
```

Pass the signal and the next power of 2 to the `fft` function.

```
y = fft(x,n);
```

See Also

`fft` | `log2` | `pow2`

nnz

Purpose

Number of nonzero matrix elements

Syntax

```
n = nnz(X)
```

Description

`n = nnz(X)` returns the number of nonzero elements in matrix `X`.

The density of a sparse matrix is `nnz(X)/prod(size(X))`.

Examples

The matrix

```
w = sparse(wilkinson(21));
```

is a tridiagonal matrix with `20` nonzeros on each of three diagonals, so `nnz(w) = 60`.

See Also

`find` | `isa` | `nonzeros` | `nzmax` | `size` | `whos`

noanimate

Purpose

Change `EraseMode` of all objects to `normal`

Note `noanimate` will be removed in a future release.

Syntax

```
noanimate(state,fig_handle)
noanimate(state)
```

Description

`noanimate(state,fig_handle)` sets the `EraseMode` of all image, line, patch, surface, and text graphics objects in the specified figure to `normal`. `state` can be the following strings:

- `'save'` — Set the values of the `EraseMode` properties to `normal` for all the appropriate objects in the designated figure.
- `'restore'` — Restore the `EraseMode` properties to the previous values (i.e., the values before calling `noanimate` with the `'save'` argument).

`noanimate(state)` operates on the current figure.

`noanimate` is useful if you want to print the figure to a TIFF or JPEG format.

See Also

`print`

Purpose

Nonzero matrix elements

Syntax

```
s = nonzeros(A)
```

Description

`s = nonzeros(A)` returns a full column vector of the nonzero elements in `A`, ordered by columns.

This gives the `s`, but not the `i` and `j`, from `[i,j,s] = find(A)`. Generally,

```
length(s) = nnz(A) <= nzmax(A) <= prod(size(A))
```

See Also

`find` | `isa` | `nnz` | `nzmax` | `size` | `whos`

Purpose

Vector and matrix norms

Syntax

```
n = norm(X,2)
n = norm(X)
n = norm(X,1)
n = norm(X,Inf)
n = norm(X,'fro')
n = norm(v,p)
n = norm(v,Inf)
n = norm(v,-Inf)
```

Description

The `norm` function calculates several different types of matrix and vector norms. If the input is a vector or a matrix:

`n = norm(X,2)` returns the 2-norm of `X`.

`n = norm(X)` is the same as `n = norm(X,2)`.

`n = norm(X,1)` returns the 1-norm of `X`.

`n = norm(X,Inf)` returns the infinity norm of `X`.

`n = norm(X,'fro')` returns the Frobenius norm of `X`.

In addition, when the input is a vector `v`:

`n = norm(v,p)` returns the *p*-norm of `v`. The *p*-norm is `sum(abs(v).^p)^(1/p)`.

`n = norm(v,Inf)` returns the largest element of `abs(v)`.

`n = norm(v,-Inf)` returns the smallest element of `abs(v)`.

Tips

By convention, `norm` returns `NaN` if the input matrix `X` or input vector `v` contains `NaN` values.

See Also

`cond` | `condest` | `hypot` | `normest` | `rcond`

normest

Purpose

2-norm estimate

Syntax

```
nrm = normest(S)
nrm = normest(S,tol)
[nrm,count] = normest(...)
```

Description

This function is intended primarily for sparse matrices, although it works correctly and may be useful for large, full matrices as well.

`nrm = normest(S)` returns an estimate of the 2-norm of the matrix `S`.

`nrm = normest(S,tol)` uses relative error `tol` instead of the default tolerance `1.e-6`. The value of `tol` determines when the estimate is considered acceptable.

`[nrm,count] = normest(...)` returns an estimate of the 2-norm and also gives the number of power iterations used.

Algorithms

The power iteration involves repeated multiplication by the matrix `S` and its transpose, `S'`. The iteration is carried out until two successive estimates agree to within the specified relative tolerance.

See Also

`cond` | `condest` | `norm` | `rcond` | `svd`

Purpose

Find logical NOT of array or scalar input

Syntax

```
~A
not(A)
```

Description

`~A` performs a logical NOT of input array `A`, and returns an array containing elements set to either logical 1 (`true`) or logical 0 (`false`). An element of the output array is set to 1 if the input array contains a zero value element at that same array location. Otherwise, that element is set to 0.

The input of the expression can be an array or can be a scalar value. If the input is an array, then the output is an array of the same dimensions. If the input is scalar, then the output is scalar.

`not(A)` is called for the syntax `~A` when `A` is an object.

Examples

If matrix `A` is

```
     0    29     0    36     0
    23    34    35     0    39
     0    24    31    27     0
     0    29     0     0    34
```

then

```
~A
ans =
     1     0     1     0     1
     0     0     0     1     0
     1     0     0     0     1
     1     0     1     1     0
```

See Also

`bitcmp` | `and` | `or` | `xor` | `any` | `all`

How To

- "Logical Operators"
- "Bit-Wise Functions"

notebook

Purpose

Open MATLAB Notebook in Microsoft Word software (on Microsoft Windows platforms)

Syntax

```
notebook
notebook('filename')
notebook('-setup')
```

Description

`notebook` starts Microsoft Word software and creates a new MATLAB Notebook titled `Document 1`.

`notebook('filename')` starts Microsoft Word and opens the notebook `filename`, where `filename` is either in the MATLAB current folder or is a full path. If `filename` does not exist, MATLAB creates a new notebook titled `filename`. If the file name extension is not specified, MATLAB assumes `.doc`.

`notebook('-setup')` runs an interactive setup function for MATLAB Notebook. It copies the notebook template, `m-book.dot`, to the Microsoft Word template folder, whose location MATLAB automatically determines from the Windows system registry. Upon completion, MATLAB displays a message indicating whether or not the setup was successful.

How To

- “Create a MATLAB Notebook with Microsoft Word”
- “Publishing MATLAB Code”

Purpose

Notify listeners that event is occurring

Syntax

```
notify(Hobj,'EventName')
notify(Hobj,'EventName',data)
```

Description

`notify(Hobj,'EventName')` notifies listeners that the specified event is taking place on the specified handle objects.

`notify(Hobj,'EventName',data)` includes user-defined event data.

Input Arguments

`Hobj`
: Array of handle objects triggering the specified event.

`EventName`
: Name of the event.

`data`
: An `event.EventData` object encapsulating information about the event. You can define custom event data by subclassing `event.EventData` and passing an instance of your subclass as the `data` argument. See "Defining Event-Specific Data" for information on defining event data.

See Also

See "Events and Listeners — Syntax and Techniques"

`handle`, `addlistener`

now

Purpose

Current date and time as serial date number

Syntax

```
t = now
```

Description

`t = now` returns the current date and time as a serial date number. To return the time only, use `rem(now,1)`. To return the date only, use `floor(now)`.

Examples

```
t1 = now, t2 = rem(now,1)

t1 =

   7.2908e+05


t2 =

    0.4013
```

See Also

`clock` | `date` | `datenum`

Purpose

Real nth root of real numbers

Syntax

```
y = nthroot(X, n)
```

Description

`y = nthroot(X, n)` returns the real `n`th root of the elements of `X`. Both `X` and `n` must be real and `n` must be a scalar. If `X` has negative entries, `n` must be an odd integer.

Examples

```
nthroot(-2, 3)
```

returns the real cube root of `-2`.

```
ans =

   -1.2599
```

By comparison,

```
(-2)^(1/3)
```

returns a complex cube root of `-2`.

```
ans =

   0.6300 + 1.0911i
```

See Also

`power` | `sqrt`

Purpose

Null space

Syntax

```
Z = null(A)
Z = null(A,'r')
```

Description

`Z = null(A)` is an orthonormal basis for the null space of `A` obtained from the singular value decomposition. That is, `A*Z` has negligible elements, `size(Z,2)` is the nullity of `A`, and `Z'*Z = I`.

`Z = null(A,'r')` is a "rational" basis for the null space obtained from the reduced row echelon form. `A*Z` is zero, `size(Z,2)` is an estimate for the nullity of `A`, and, if `A` is a small matrix with integer elements, the elements of the reduced row echelon form (as computed using `rref`) are ratios of small integers.

The orthonormal basis is preferable numerically, while the rational basis may be preferable pedagogically.

Examples

Example 1

Compute the orthonormal basis for the null space of a matrix A.

```
A = [1     2     3
     1     2     3
     1     2     3];

Z = null(A);
A*Z

ans =
   1.0e-015 *
      0.2220    0.2220
      0.2220    0.2220
      0.2220    0.2220

Z'*Z

ans =
```

```
    1.0000   -0.0000
   -0.0000    1.0000
```

Example 2

Compute the 1-norm of the matrix A*Z and determine that it is within a small tolerance.

```
norm(A*Z,1) < 1e-12
ans =
    1
```

Example 3

Compute the rational basis for the null space of the same matrix A.

```
ZR = null(A,'r')

ZR =
    -2    -3
     1     0
     0     1

A*ZR

ans =

     0     0
     0     0
     0     0
```

See Also

orth | rank | rref | svd

num2cell

Purpose

Convert array to cell array with consistently sized cells

Syntax

```
C = num2cell(A)
C = num2cell(A,dim)
```

Description

`C = num2cell(A)` converts array `A` into cell array `C` by placing each element of `A` into a separate cell in `C`. Array `A` need not be numeric.

`C = num2cell(A,dim)` splits the contents of `A` into separate cells of `C`, where `dim` specifies which dimensions of `A` to include in each cell. `dim` can be a scalar or a vector of dimensions. For example, if `A` has 2 rows and 3 columns, then:

- `num2cell(A,1)` creates a 1-by-3 cell array `C`, where each cell contains a 2-by-1 column of `A`.
- `num2cell(A,2)` creates a 2-by-1 cell array `C`, where each cell contains a 1-by-3 row of `A`.
- `num2cell(A,[1 2])` creates a 1-by-1 cell array `C`, where the cell contains the entire array `A`.

Input Arguments

A - Input

any type of multidimensional array

Input, specified as any type of multidimensional array.

Data Types

`single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | struct | cell | function_handle`

dim - Dimension of A

positive integer | positive vector of integers

Dimension of `A`, specified as a positive integer or a vector of positive integers. `dim` must be between 1 and `ndims(A)`.

Elements need not be in numeric order. However, `num2cell` permutes the dimensions of the arrays in each cell of `C` to match the order of the specified dimensions.

Data Types
`double`

Output Arguments

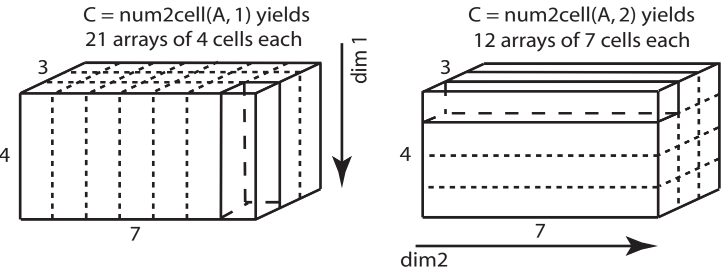
C - Resulting array

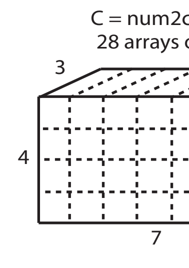
cell array

Resulting array, returned as a cell array. The size of `C` depends on the size of `A` and the values of `dim`.

- If `dim` is not specified, then `C` is the same size as `A`.
- If `dim` is a scalar, then `C` contains `numel(A)/size(A,dim)` cells. If `dim` is 1 or 2, then each cell contains a column or row vector, respectively. If `dim` > 2, then each cell contains an array whose `dim`th dimensional length is `size(A,dim)`, and whose other dimensions are all singletons.

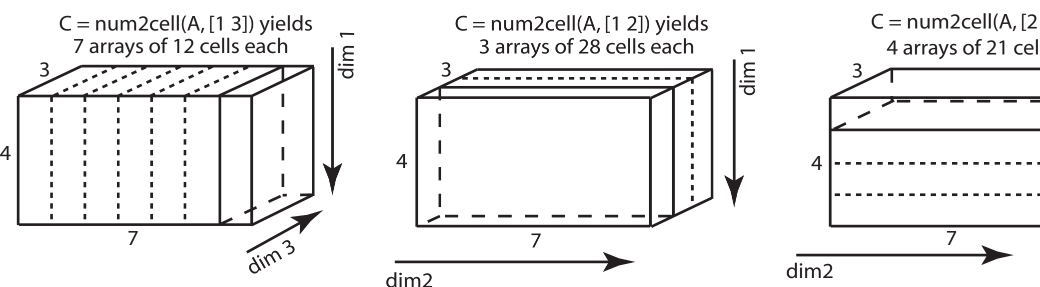
  For example, given a 4-by-7-by-3 array, `A`, this figure shows how `num2cell` creates cells corresponding to `dim` values of `1`, `2`, and `3`.





- If `dim` is a vector containing `N` values, then `C` has `numel(A)/prod([size(A,dim(1)),...,size(A,vdim(N))])` cells. Each cell contains an array whose `dim(i)`th dimension has a length of `size(A,dim(i))` and whose other dimensions are singletons.

  For example, given a 4-by-7-by-3 array, you can specify `dim` as an positive integer vector to create cell arrays of different dimensions.



Examples

Convert Arrays to Cell Array

Place all elements of a numeric array into separate cells.

```
a = magic(3)
c = num2cell(a)
```

```
a =

     8     1     6
     3     5     7
     4     9     2


c =

    [8]    [1]    [6]
    [3]    [5]    [7]
    [4]    [9]    [2]
```

Place individual letters of a word into separate cells of an array.

```
a = ['four';'five';'nine']
c = num2cell(a)
```

```
a =

four
```

```
five
nine


c = 

    'f'    'o'    'u'    'r'
    'f'    'i'    'v'    'e'
    'n'    'i'    'n'    'e'
```

Create Cell Array of Numeric Arrays

Generate a 4-by-3-by-2 numeric array, and then create a 1-by-3-by-2 cell array of 4-by-1 column vectors.

```
A = reshape(1:12,4,3);
A(:,:,2) = A*10
C = num2cell(A,1)
```

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

     1     5     9
     2     6    10
     3     7    11
     4     8    12


A(:,:,2) =

    10    50    90
    20    60   100
    30    70   110
    40    80   120


C(:,:,1) = 

    [4x1 double]    [4x1 double]    [4x1 double]
```

```
C(:,:,2) =

    [4x1 double]    [4x1 double]    [4x1 double]
```

Each 4-by-1 vector contains elements from along the *first* dimension of A:

```
C{1}

ans =

     1
     2
     3
     4
```

Create a 4-by-1-by-2 cell array of 1-by-3 numeric arrays.

```
C = num2cell(A,2)

C(:,:,1) =

    [1x3 double]
    [1x3 double]
    [1x3 double]
    [1x3 double]


C(:,:,2) =

    [1x3 double]
    [1x3 double]
    [1x3 double]
    [1x3 double]
```

Each 1-by-3 row vector contains elements from along the *second* dimension of A:

```
C{1}

ans =

     1     5     9
```

Finally, create a 4-by-3 cell array of 1-by-1-by-2 numeric arrays.

```
C = num2cell(A,3)

C =

    [1x1x2 double]    [1x1x2 double]    [1x1x2 double]
    [1x1x2 double]    [1x1x2 double]    [1x1x2 double]
    [1x1x2 double]    [1x1x2 double]    [1x1x2 double]
    [1x1x2 double]    [1x1x2 double]    [1x1x2 double]
```

Each 1-by-1-by-2 vector contains elements from along the *third* dimension of A:

```
C{1}

ans(:,:,1) =

     1


ans(:,:,2) =

    10
```

Combine Across Multiple Dimensions

Create a cell array by combining elements into numeric arrays along several dimensions.

```
a = reshape(1:12,4,3);
a(:,:,2) = a*10
c = num2cell(A,[1 3])
```

```
a(:,:,1) =

     1     5     9
     2     6    10
     3     7    11
     4     8    12


a(:,:,2) =

    10    50    90
    20    60   100
    30    70   110
    40    80   120


c =

    [4x1x2 double]    [4x1x2 double]    [4x1x2 double]
```

Each 4-by-1-by-2 array contains elements from along the first and third dimension of `A`:

```
c{1}
```

```
ans(:,:,1) =

     1
     2
     3
     4


ans(:,:,2) =

    10
    20
```

```
    30
    40

c = num2cell(A,[2 3])

c =

    [1x3x2 double]
    [1x3x2 double]
    [1x3x2 double]
    [1x3x2 double]
```

See Also

cat | mat2cell | cell2mat

num2hex

Purpose

Convert singles and doubles to IEEE hexadecimal strings

Syntax

```
num2hex(X)
```

Description

If `X` is a single or double precision array with n elements, `num2hex(X)` is an n-by-8 or n-by-16 char array of the hexadecimal floating-point representation. The same representation is printed with format `hex`.

Examples

```
num2hex([1 0 0.1 -pi Inf NaN])
```

returns

```
ans =

3ff0000000000000
0000000000000000
3fb999999999999a
c00921fb54442d18
7ff0000000000000
fff8000000000000
num2hex(single([1 0 0.1 -pi Inf NaN]))
```

returns

```
ans =

3f800000
00000000
3dcccccd
c0490fdb
7f800000
ffc00000
```

See Also

`hex2num` | `dec2hex` | `format`

Purpose

Convert number to string

Syntax

```
str = num2str(A)
str = num2str(A,precision)
str = num2str(A,formatSpec)
```

Description

`str = num2str(A)` converts array `A` into a string representation `str`. The output format depends upon the magnitude of the original values and sometimes includes exponents. `num2str` is useful for labeling and titling plots with numeric values.

`str = num2str(A,precision)` returns a string representation with the maximum number of digits specified by `precision`.

`str = num2str(A,formatSpec)` applies a format specified by `formatSpec` to all elements of `A`.

Input Arguments

A - Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of real or complex numbers.

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

precision - Numeric precision

positive integer

Numeric precision of the output string, specified as a positive integer. This is the maximum number of digits in the output string.

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

formatSpec - Format of the output fields
string

Format of the output fields, specified as a string.

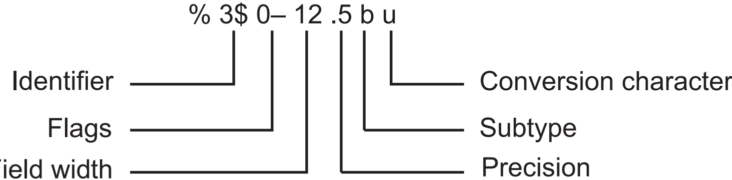
The string can include a percent sign followed by a conversion character. The following table lists the available conversion characters and subtypes.

Value Type	Conversion	Details
Integer, signed	`%d` or `%i`	Base 10
Integer, unsigned	`%u`	Base 10
	`%o`	Base 8 (octal)
	`%x`	Base 16 (hexadecimal), lowercase letters `a`–`f`
	`%X`	Same as `%x`, uppercase letters `A`–`F`
Floating-point number	`%f`	Fixed-point notation
	`%e`	Exponential notation, such as `3.141593e+00`
	`%E`	Same as `%e`, but uppercase, such as `3.141593E+00`
	`%g`	The more compact of `%e` or `%f`, with no trailing zeros
	`%G`	The more compact of `%E` or `%f`, with no trailing zeros

	`%bx` or `%bX`
	`%bo`
	`%bu`
		Double-precision hexadecimal, octal, or decimal value
		Example: `%bx` prints `pi` as `400921fb54442d18`

Value Type	Conversion	Details
	%tx or %tX %to %tu	Single-precision hexadecimal, octal, or decimal value Example: %tx prints pi as 40490fdb
Characters	%c	Single character
	%s	String of characters

The string can include optional operators, which appear in the following order (includes spaces for clarity):



Optional operators include:

- Identifier

  Order for processing inputs. Use the syntax *n*$, where *n* represents the position of the value in the input list.

  For example, '%3$s %2$s %1$s %2$s' prints inputs 'A', 'B', 'C' as follows: C B A B.

- Flags

' '	Left-justify. Example: %-5.2f
'+'	Print sign character (+) for positive values. Example: %+5.2f
' '	Pad to field width with spaces before the value. Example: % 5.2f

`'0'`	Pad to field width with zeros. Example: `%05.2f`
`'#'`	Modify selected numeric conversions: • For `%o`, `%x`, or `%X`, print `0`, `0x`, or `0X` prefix. - For `%f`, `%e`, or `%E`, print decimal point even when precision is 0. - For `%g` or `%G`, do not remove trailing zeros or decimal point. Example: `%#5.0f`

- Field width

  Minimum number of characters to print. Can be a number, or an asterisk (`*`) to refer to an argument in the input list. For example, the input list `('%12d', intmax)` is equivalent to `('%*d', 12, intmax)`.

- Precision

For `%f`, `%e`, or `%E`:	Number of digits to the right of the decimal point. Example: `'%6.4f'` prints `pi` as `'3.1416'`
For `%g` or `%G`	Number of significant digits. Example: `'%6.4g'` prints `pi` as `' 3.142'`

  Can be a number, or an asterisk (`*`) to refer to an argument in the input list. For example, the input list `('%6.4f', pi)` is equivalent to `('%*.*f', 6, 4, pi)`.

The string can also include combinations of the following:

- Literal text to print. To print a single quotation mark, include `''` in `formatSpec`.
- Control characters, including:

`%%`	Percent character
`\\`	Backslash
`\a`	Alarm
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\xN`	Character whose ASCII code is the hexadecimal number, `N`
`\N`	Character whose ASCII code is the octal number, `N`

The following limitations apply to conversions:

- Numeric conversions print only the real component of complex numbers.
- If you specify a conversion that does not fit the data, such as a string conversion for a numeric value, MATLAB overrides the specified conversion, and uses `%e`.
- If you apply a string conversion (`%s`) to integer values, MATLAB converts values that correspond to valid character codes to characters. For example, `'%s'` converts `[65 66 67]` to `ABC`.

Output Arguments

str - String representation of input array

character array

String representation of the input array, `A`, returned as a character array.

Algorithms

`num2str` trims any leading spaces from a string, even when `formatSpec` includes a space character flag. For example, `num2str(42.67,'% 10.2f')` returns a 1-by-5 character array `'42.67'`.

Examples

Default Conversions of Floating-Point Values

Convert `pi` and `eps` to text strings.

```
str = num2str(pi)

str =
3.1416

str = num2str(eps)

str =
2.2204e-16
```

Specifying Precision

Specify the number of significant digits for floating-point values.

```
A = gallery('normaldata',[2,2],0);
precision = 3;
str = num2str(A,precision)

str =
-0.433     0.125
 -1.67     0.288
```

Specifying Formatting

Specify the width, precision, and other formatting for an array of floating-point values.

```
A = gallery('uniformdata',[2,3],0) * 9999;
formatSpec = '%10.5e\n';
str = num2str(A,formatSpec)

str =
9.50034e+03
```

```
6.06782e+03
8.91210e+03
2.31115e+03
4.85934e+03
7.62021e+03
```

The format `'%10.5e'` prints each value in exponential format with five decimal places, and `'\n'` prints a new line character.

See Also

`cast` | `int2str` | `mat2str` | `sprintf` | `str2num`

Tiff.numberOfStrips

Purpose

Total number of strips in image

Syntax

`numStrips = tiffobj.numberOfStrips()`

Description

`numStrips = tiffobj.numberOfStrips()` returns the total number of strips in the image.

Examples

Open a `Tiff` object and determine the number of strips in the image. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path:

```
t = Tiff('myfile.tif', 'r');
%
% Check if image has a stripped organization
if ~t.isTiled()
  nStrips = t.numberOfStrips();
end
```

References

This method corresponds to the `TIFFNumberOfStrips` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff.numberOfTiles` | `Tiff.isTiled`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

Purpose

Total number of tiles in image

Syntax

`numTiles = tiffobj.numberOfTiles()`

Description

`numTiles = tiffobj.numberOfTiles()` returns the total number of tiles in the image.

Examples

Open a `Tiff` object and determine the number of tiles in the image. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path:

```
t = Tiff('myfile.tif', 'r');
%
% Check if image has a tiled organization
if t.isTiled()
  nTiles = t.numberOfTiles();
end
```

References

This method corresponds to the `TIFFNumberOfTiles` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff.numberOfStrips` | `Tiff.isTiled`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

numel

Purpose

Number of array elements

Syntax

```
n = numel(A)
```

Description

`n = numel(A)` returns the number of elements, `n`, in array `A`.

Input Arguments

A - Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. This includes character, structure, cell, and object arrays.

Data Types

`single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | struct | cell | function_handle`

Complex Number Support: Yes

Examples

Number of Elements in 3-D Matrix

Create a 4-by-4-by-2 matrix.

```
A = magic(4);
A(:,:,2) = A'

A(:,:,1) =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

A(:,:,2) =
    16     5     9     4
     2    11     7    14
     3    10     6    15
    13     8    12     1
```

`numel` counts 32 elements in the matrix.

```
n = numel(A)
```

```
n =

    32
```

Number of Elements in Cell Array of Strings

Create a cell array of strings.

```
A = {'dog','cat','fish','horse'};
```

`numel` counts 4 string elements in the array.

```
n = numel(A)
```

```
n =

     4
```

See Also

`prod` | `size` | `subsref`

Concepts

- "Caution When Customizing Classes"

nzmax

Purpose

Amount of storage allocated for nonzero matrix elements

Syntax

```
n = nzmax(S)
```

Description

`n = nzmax(S)` returns the amount of storage allocated for nonzero elements.

If `S` is a sparse matrix...	`nzmax(S)` is the number of storage locations allocated for the nonzero elements in `S`.
If `S` is a full matrix...	`nzmax(S) = prod(size(S))`.

Often, `nnz(S)` and `nzmax(S)` are the same. But if `S` is created by an operation which produces fill-in matrix elements, such as sparse matrix multiplication or sparse LU factorization, more storage may be allocated than is actually required, and `nzmax(S)` reflects this. Alternatively, `sparse(i,j,s,m,n,nzmax)` or its simpler form, `spalloc(m,n,nzmax)`, can set `nzmax` in anticipation of later fill-in.

See Also

`find` | `isa` | `nnz` | `nonzeros` | `size` | `whos`

Purpose

Solve fully implicit differential equations, variable order method

Syntax

```
[T,Y] = ode15i(odefun,tspan,y0,yp0)
[T,Y] = ode15i(odefun,tspan,y0,yp0,options)
[T,Y,TE,YE,IE] = ode15i(odefun,tspan,y0,yp0,options...)
sol = ode15i(odefun,[t0 tfinal],y0,yp0,...)
```

Arguments

The following table lists the input arguments for `ode15i`.

`odefun`	A function handle that evaluates the left side of the differential equations, which are of the form $f(t,y,y') = 0$.
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.
`y0, yp0`	Vectors of initial conditions for y and y' respectively.
`options`	Optional integration argument created using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for `ode15i`.

`T`	Column vector of time points
`Y`	Solution array. Each row in `y` corresponds to the solution at a time returned in the corresponding row of `t`.

Description

`[T,Y] = ode15i(odefun,tspan,y0,yp0)` with `tspan = [t0 tf]` integrates the system of differential equations $f(t,y,y') = 0$ from time `t0` to `tf` with initial conditions `y0` and `yp0`. `odefun` is a function handle. Function `ode15i` solves ODEs and DAEs of index 1. The initial conditions must be consistent, meaning that $f(t0,y0,yp0) = 0$. You can use the function `decic` to compute consistent initial conditions close to guessed values. Function `odefun(t,y,yp)`, for a scalar `t` and column vectors `y` and `yp`, must return a column vector corresponding to $f(t,y,y')$. Each row in the solution array `Y` corresponds to a time returned in the

column vector `T`. To obtain solutions at specific times `t0,t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

"Parameterizing Functions" explains how to provide additional parameters to the function `odefun`, if necessary.

`[T,Y] = ode15i(odefun,tspan,y0,yp0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used options include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components `1e-6` by default). See `odeset` for details.

`[T,Y,TE,YE,IE] = ode15i(odefun,tspan,y0,yp0,options...)` with the `'Events'` property in `options` set to a function `events`, solves as above while also finding where functions of (t,y,y'), called event functions, are zero. The function `events` is of the form `[value,isterminal,direction] = events(t,y,yp) and` includes the necessary event functions. Code the function `events` so that the `i`th element of each output vector corresponds to the `i`th event. For the `i`th event function in `events`:

- `value(i)` is the value of the function.
- `isterminal(i) = 1` if the integration is to terminate at a zero of this event function and `0` otherwise.
- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Output `TE` is a column vector of times at which events occur. Rows of `YE` are the corresponding solutions, and indices in vector `IE` specify which event occurred. See "Integrator Options" in the MATLAB Mathematics documentation for more information.

`sol = ode15i(odefun,[t0 tfinal],y0,yp0,...)` returns a structure that can be used with `deval` to evaluate the solution at any point between `t0` and `tfinal`. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver. If you specify the `Events` option and a terminal event is detected, `sol.x(end)` contains the end of the step at which the event occurred.
`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

Options

`ode15i` accepts the following parameters in `options`. For more information, see `odeset` and Changing ODE Integration Properties in the MATLAB Mathematics documentation.

Error control	`RelTol`, `AbsTol`, `NormControl`
Solver output	`OutputFcn`, `OutputSel`, `Refine`, `Stats`
Event location	`Events`
Step size	`MaxStep`, `InitialStep`
Jacobian matrix	`Jacobian`, `JPattern`, `Vectorized`

Solver Output

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen. By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

Jacobian Matrices

The Jacobian matrices $\partial f/\partial y$ and $\partial f/\partial y'$ are critical to reliability and efficiency. You can provide these matrices as one of the following:

- Function of the form `[dfdy,dfdyp] = FJAC(t,y,yp)` that computes the Jacobian matrices. If `FJAC` returns an empty matrix `[]` for either `dfdy` or `dfdyp`, then `ode15i` approximates that matrix by finite differences.
- Cell array of two constant matrices `{dfdy,dfdyp}`, either of which could be empty.

Use `odeset` to set the `Jacobian` option to the function or cell array. If you do not set the `Jacobian` option, `ode15i` approximates both Jacobian matrices by finite differences.

For `ode15i`, `Vectorized` is a two-element cell array. Set the first element to `'on'` if `odefun(t,[y1,y2,...],yp)` returns `[odefun(t,y1,yp),odefun(t,y2,yp),...]`. Set the second element to `'on'` if `odefun(t,y,[yp1,yp2,...])` returns `[odefun(t,y,yp1),odefun(t,y,yp2),...]`. The default value of `Vectorized` is `{'off','off'}`.

For `ode15i`, `JPattern` is also a two-element sparse matrix cell array. If $\partial f/\partial y$ or $\partial f/\partial y'$ is a sparse matrix, set `JPattern` to the sparsity patterns, `{SPDY,SPDYP}`. A sparsity pattern of $\partial f/\partial y$ is a sparse matrix `SPDY` with `SPDY(i,j) = 1` if component `i` of `f(t,y,yp)` depends on component `j` of `y`, and `0` otherwise. Use `SPDY = []` to indicate that $\partial f/\partial y$ is a full matrix. Similarly for $\partial f/\partial y'$ and `SPDYP`. The default value of `JPattern` is `{[],[]}`.

Examples

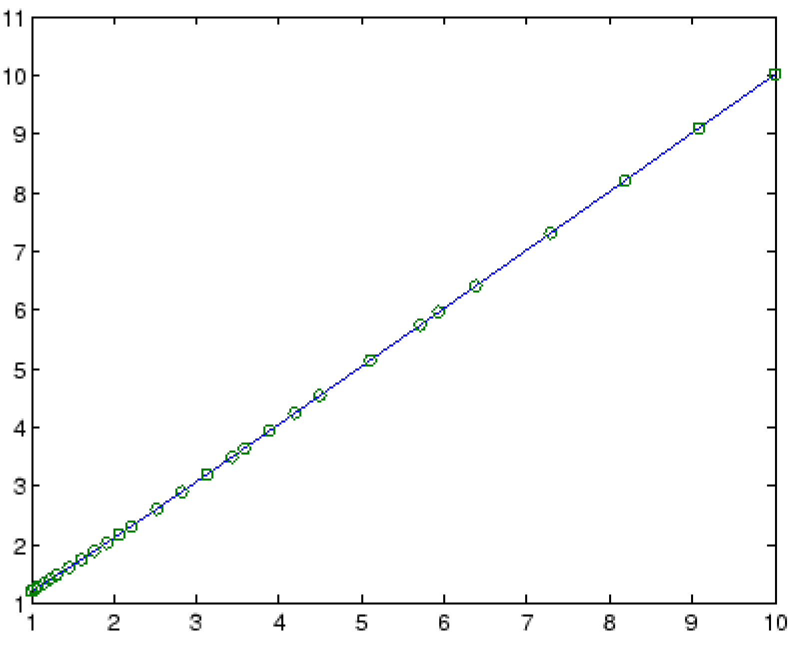
Example 1

This example uses a helper function `decic` to hold fixed the initial value for $y(t_0)$ and compute a consistent initial value for $y'(t_0)$ for the Weissinger implicit ODE. The Weissinger function evaluates the residual of the implicit ODE.

```
t0 = 1;
y0 = sqrt(3/2);
yp0 = 0;
[y0,yp0] = decic(@weissinger,t0,y0,1,yp0,0);
```

The example uses `ode15i` to solve the ODE, and then plots the numerical solution against the analytical solution.

```
[t,y] = ode15i(@weissinger,[1 10],y0,yp0);
ytrue = sqrt(t.^2 + 0.5);
plot(t,y,t,ytrue,'o');
```



Other Examples

The files, `ihb1dae.m` and `iburgersode.m`, are examples of implicit ODEs.

See Also

`decic` | `deval` | `odeget` | `odeset` | `function_handle` | `ode45` | `ode23` | `ode113` | `ode15s` | `ode23s` | `ode23t` | `ode23tb`

Purpose

Solve stiff differential equations and DAEs; variable order method

Syntax

```
[T,Y] = solver(odefun,tspan,y0)
[T,Y] = solver(odefun,tspan,y0,options)
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
sol = solver(odefun,[t0 tf],y0...)
```

This page contains an overview of the solver functions: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. You can call any of these solvers by substituting the placeholder, *`solver`*, with any of the function names.

Arguments

The following table describes the input arguments to the solvers.

`odefun`	A function handle that evaluates the right side of the differential equations. All solvers solve systems of equations in the form $y' = f(t,y)$ or problems that involve a mass matrix, $M(t,y)y' = f(t,y)$. The `ode23s` solver can solve only equations with constant mass matrices. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`. For `tspan` vectors with two elements `[t0 tf]`, the solver returns the solution evaluated at every integration step. For `tspan` vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.

	Specifying `tspan` with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from `tspan(1)` to `tspan(end)`. All solvers in the ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in `tspan`, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points. Specifying `tspan` with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management.
`y0`	A vector of initial conditions.
`options`	Structure of optional parameters that change the default integration properties. This is the fourth input argument. `[t,y] = solver(odefun,tspan,y0,options)` You can create options using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for the solvers.

`T`	Column vector of time points.
`Y`	Solution array. Each row in `Y` corresponds to the solution at a time returned in the corresponding row of `T`.
`TE`	The time at which an event occurs.
`YE`	The solution at the time of the event.

`IE`	The index `i` of the event function that vanishes.
`sol`	Structure to evaluate the solution.

Description

`[T,Y] = solver(odefun,tspan,y0)` with `tspan = [t0 tf]` integrates the system of differential equations $y' = f(t,y)$ from time `t0` to `tf` with initial conditions `y0`. The first input argument, `odefun`, is a function handle. The function, `f = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `f` corresponding to $f(t,y)$. Each row in the solution array `Y` corresponds to a time returned in column vector `T`. To obtain solutions at the specific times `t0`, `t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`[T,Y] = solver(odefun,tspan,y0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used properties include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components are `1e-6` by default). If certain components of the solution must be nonnegative, use the `odeset` function to set the `NonNegative` property to the indices of these components. See `odeset` for details.

`[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)` solves as above while also finding where functions of (t,y), called event functions, are zero. For each event function, you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, e.g., `events` or `@events`, and creating a function `[value,isterminal,direction] = events(t,y)`. For the `i`th event function in `events`,

- `value(i)` is the value of the function.
- `isterminal(i) = 1`, if the integration is to terminate at a zero of this event function and `0` otherwise.

- direction(i) = 0 if all zeros are to be computed (the default), +1 if only the zeros where the event function increases, and -1 if only the zeros where the event function decreases.

Corresponding entries in TE, YE, and IE return, respectively, the time at which an event occurs, the solution at the time of the event, and the index i of the event function that vanishes.

sol = *solver*(odefun,[t0 tf],y0...) returns a structure that you can use with deval to evaluate the solution at any point on the interval [t0,tf]. You must pass odefun as a function handle. The structure sol always includes these fields:

sol.x	Steps chosen by the solver.
sol.y	Each column sol.y(:,i) contains the solution at sol.x(i).
sol.solver	Solver name.

If you specify the Events option and events are detected, sol also includes these fields:

sol.xe	Points at which events, if any, occurred. sol.xe(end) contains the exact point of a terminal event, if any.
sol.ye	Solutions that correspond to events in sol.xe.
sol.ie	Indices into the vector returned by the function specified in the Events option. The values indicate which event the solver detected.

If you specify an output function as the value of the OutputFcn property, the solver calls it with the computed solution after each time step. Four output functions are provided: odeplot, odephas2, odephas3, odeprint. When you call the solver with no output arguments, it calls the default odeplot to plot the solution as it is computed. odephas2 and odephas3 produce two- and three-dimensional phase plane plots, respectively. odeprint displays the solution components on the screen.

By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

For the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, the Jacobian matrix $\partial f/\partial y$ is critical to reliability and efficiency. Use `odeset` to set `Jacobian` to `@FJAC` if `FJAC(T,Y)` returns the Jacobian $\partial f/\partial y$ or to the matrix $\partial f/\partial y$ if the Jacobian is constant. If the `Jacobian` property is not set (the default), $\partial f/\partial y$ is approximated by finite differences. Set the `Vectorized` property 'on' if the ODE function is coded so that `odefun(T,[Y1,Y2 ...])` returns `[odefun(T,Y1),odefun(T,Y2) ...]`. If $\partial f/\partial y$ is a sparse matrix, set the `JPattern` property to the sparsity pattern of $\partial f/\partial y$, i.e., a sparse matrix `S` with `S(i,j) = 1` if the `i`th component of $f(t,y)$ depends on the `j`th component of y, and 0 otherwise.

The solvers of the ODE suite can solve problems of the form $M(t,y)y' = f(t,y)$, with time- and state-dependent mass matrix M. (The `ode23s` solver can solve only equations with constant mass matrices.) If a problem has a mass matrix, create a function `M = MASS(t,y)` that returns the value of the mass matrix, and use `odeset` to set the `Mass` property to `@MASS`. If the mass matrix is constant, the matrix should be used as the value of the `Mass` property. Problems with state-dependent mass matrices are more difficult:

- If the mass matrix does not depend on the state variable y and the function `MASS` is to be called with one input argument, `t`, set the `MStateDependence` property to '`none`'.
- If the mass matrix depends weakly on y, set `MStateDependence` to '`weak`' (the default); otherwise, set it to '`strong`'. In either case, the function `MASS` is called with the two arguments `(t,y)`.

If there are many differential equations, it is important to exploit sparsity:

- Return a sparse $M(t,y)$.

- Supply the sparsity pattern of $\partial f/\partial y$ using the `JPattern` property or a sparse $\partial f/\partial y$ using the `Jacobian` property.
- For strongly state-dependent $M(t,y)$, set `MvPattern` to a sparse matrix `S` with `S(i,j) = 1` if for any `k`, the `(i,k)` component of $M(t,y)$ depends on component `j` of y, and `0` otherwise.

If the mass matrix M is singular, then $M(t,y)y' = f(t,y)$ is a system of differential algebraic equations. DAEs have solutions only when y_0 is consistent, that is, if there is a vector yp_0 such that $M(t_0,y_0)yp_0 = f(t_0,y_0)$. The `ode15s` and `ode23t` solvers can solve DAEs of index 1 provided that `y0` is sufficiently close to being consistent. If there is a mass matrix, you can use `odeset` to set the `MassSingular` property to `'yes'`, `'no'`, or `'maybe'`. The default value of `'maybe'` causes the solver to test whether the problem is a DAE. You can provide `yp0` as the value of the `InitialSlope` property. The default is the zero vector. If a problem is a DAE, and `y0` and `yp0` are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. When solving DAEs, it is very advantageous to formulate the problem so that M is a diagonal matrix (a semi-explicit DAE).

Solver	Problem Type	Order of Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.
`ode23`	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.

Solver	Problem Type	Order of Accuracy	When to Use
ode113	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.
ode15s	Stiff	Low to medium	If ode45 is slow because the problem is stiff.
ode23s	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
ode23t	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
ode23tb	Stiff	Low	If using crude error tolerances to solve stiff systems.

The algorithms used in the ODE solvers vary according to order of accuracy [6] and the type of systems (stiff or nonstiff) they are designed to solve. See “Algorithms” on page 1-3655 for more details.

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and “Integrator Options” in the MATLAB Mathematics documentation.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
RelTol, AbsTol, NormControl	√	√	√	√	√	√	√
OutputFcn, OutputSel, Refine, Stats	√	√	√	√	√	√	√
NonNegative	√	√	√	√ *	—	√ *	√ *
Events	√	√	√	√	√	√	√
MaxStep, InitialStep	√	√	√	√	√	√	√
Jacobian, JPattern, Vectorized	—	—	—	√	√	√	√
Mass MStateDependence MvPattern MassSingular	√ √ — —	√ √ — —	√ √ — —	√ √ √ √	√ — — —	√ √ √ √	√ √ √ —
InitialSlope	—	—	—	√	—	√	—
MaxOrder, BDF	—	—	—	√	—	—	—

Note You can use the NonNegative parameter with ode15s, ode23t, and ode23tb only for those problems for which there is no mass matrix.

Examples

Example 1

An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces.

$$
\begin{aligned}
y_1' &= y_2 y_3 & y_1(0) &= 0 \\
y_2' &= -y_1 y_3 & y_2(0) &= 1 \\
y_3' &= -0.51 y_1 y_2 & y_3(0) &= 1
\end{aligned}
$$

To simulate this system, create a function `rigid` containing the equations

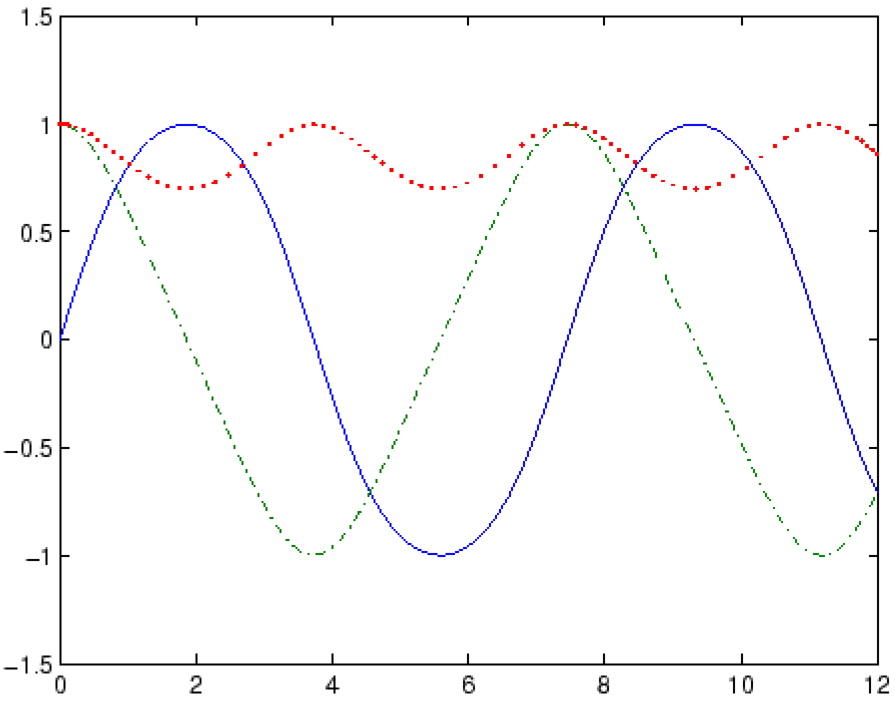
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time `0`.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[T,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

Plotting the columns of the returned array `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'.')
```



Example 2

An example of a stiff system is provided by the van der Pol equations in relaxation oscillation. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$y_1' = y_2 \qquad\qquad y_1(0) = 2$$
$$y_2' = 1000(1 - y_1^2)y_2 - y_1 \qquad y_2(0) = 0$$

To simulate this system, create a function `vdp1000` containing the equations

```
function dy = vdp1000(t,y)
```

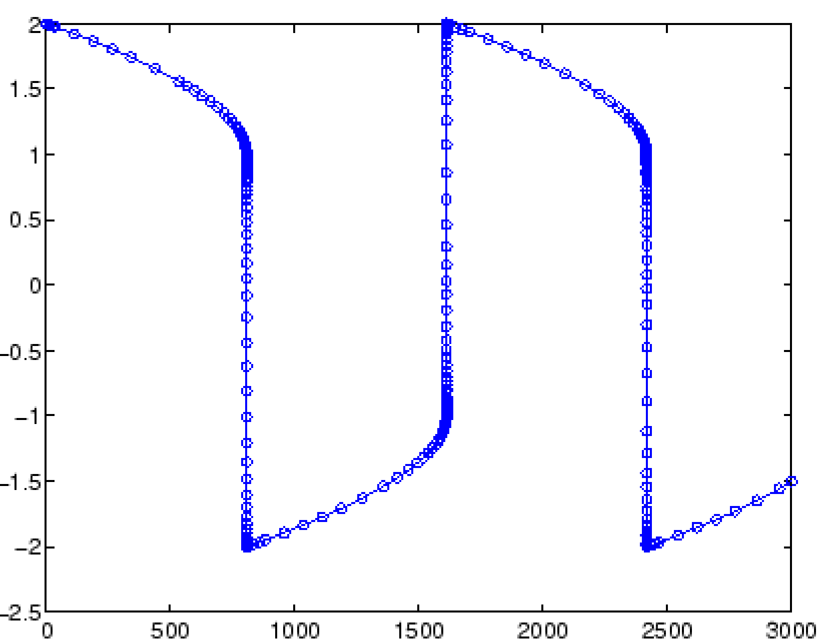
```
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

For this problem, we will use the default relative and absolute tolerances (`1e-3` and `1e-6`, respectively) and solve on a time interval of `[0 3000]` with initial condition vector `[2 0]` at time `0`.

```
[T,Y] = ode15s(@vdp1000,[0 3000],[2 0]);
```

Plotting the first column of the returned matrix `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-o')
```



Example 3

This example solves an ordinary differential equation with time-dependent terms.

Consider the following ODE, with time-dependent parameters defined only through the set of data points given in two vectors:

```
y'(t) + f(t)y(t) = g(t)
```

The initial condition is `y(0) = 0`, where the function `f(t)` is defined through the `n`-by-`1` vectors `tf` and `f`, and the function `g(t)` is defined through the `m`-by-`1` vectors `tg` and `g`.

First, define the time-dependent parameters `f(t)` and `g(t)` as the following:

```
ft = linspace(0,5,25); % Generate t for f
f = ft.^2 - ft - 3; % Generate f(t)
gt = linspace(1,6,25); % Generate t for g
g = 3*sin(gt-0.25); % Generate g(t)
```

Write a function to interpolate the data sets specified above to obtain the value of the time-dependent terms at the specified time:

```
function dydt = myode(t,y,ft,f,gt,g)
f = interp1(ft,f,t); % Interpolate the data set (ft,f) at time t
g = interp1(gt,g,t); % Interpolate the data set (gt,g) at time t
dydt = -f.*y + g; % Evalute ODE at time t
```

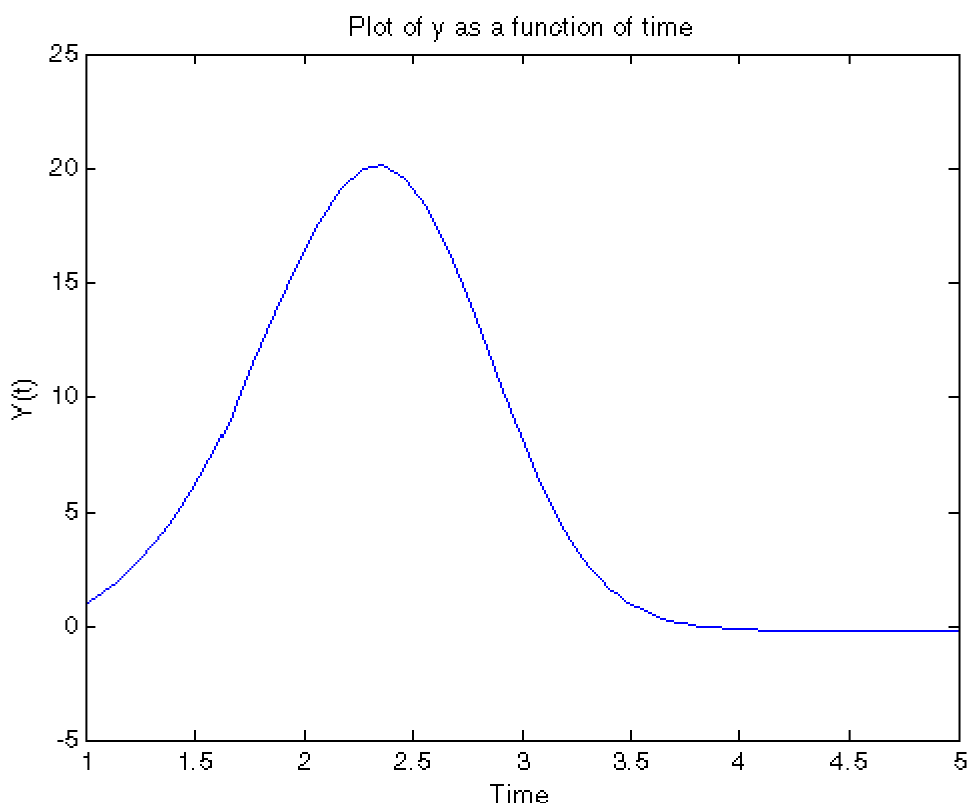
Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
Tspan = [1 5]; % Solve from t=1 to t=5
IC = 1; % y(t=0) = 1
[T Y] = ode45(@(t,y) myode(t,y,ft,f,gt,g),Tspan,IC); % Solve ODE
```

Plot the solution `y(t)` as a function of time:

```
plot(T, Y);
```

```
title('Plot of y as a function of time');
xlabel('Time'); ylabel('Y(t)');
```



Algorithms

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing `y(`t_n`)`, it needs only the solution at the immediately preceding time point, `y(`t_{n-1}`)`. In general, `ode45` is the best function to apply as a *first try* for most problems. [3]

`ode23` is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude

tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver. [2]

`ode113` is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE file function is particularly expensive to evaluate. `ode113` is a *multistep* solver — it normally needs the solutions at several preceding time points to compute the current solution. [7]

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` when `ode45` fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem. [9], [10]

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective. [9]

`ode23t` is an implementation of the trapezoidal rule using a "free" interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve DAEs. [10]

`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances. [8], [1]

References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R. Smith, "Transient Simulation of Silicon Devices and Circuits," *IEEE Trans. CAD*, 4 (1985), pp. 436–451.

[2] Bogacki, P. and L. F. Shampine, "A 3(2) pair of Runge-Kutta formulas," *Appl. Math. Letters*, Vol. 2, 1989, pp. 321–325.

[3] Dormand, J. R. and P. J. Prince, "A family of embedded Runge-Kutta formulae," *J. Comp. Appl. Math.*, Vol. 6, 1980, pp. 19–26.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, San Francisco, 1975.

[8] Shampine, L. F. and M. E. Hosea, "Analysis and Implementation of TR-BDF2," *Applied Numerical Mathematics 20*, 1996.

[9] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing,* Vol. 18, 1997, pp. 1–22.

[10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, "Solving Index-1 DAEs in MATLAB and Simulink," *SIAM Review*, Vol. 41, 1999, pp. 538–552.

See Also

`deval` | `ode15i` | `odeget` | `odeset` | `function_handle`

Purpose

Solve nonstiff differential equations; low order method

Syntax

```
[T,Y] = solver(odefun,tspan,y0)
[T,Y] = solver(odefun,tspan,y0,options)
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
sol = solver(odefun,[t0 tf],y0...)
```

This page contains an overview of the solver functions: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. You can call any of these solvers by substituting the placeholder, *`solver`*, with any of the function names.

Arguments

The following table describes the input arguments to the solvers.

`odefun`	A function handle that evaluates the right side of the differential equations. All solvers solve systems of equations in the form $y' = f(t,y)$ or problems that involve a mass matrix, $M(t,y)y' = f(t,y)$. The `ode23s` solver can solve only equations with constant mass matrices. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`. For `tspan` vectors with two elements `[t0 tf]`, the solver returns the solution evaluated at every integration step. For `tspan` vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.

	Specifying `tspan` with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from `tspan(1)` to `tspan(end)`. All solvers in the ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in `tspan`, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points. Specifying `tspan` with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management.
`y0`	A vector of initial conditions.
`options`	Structure of optional parameters that change the default integration properties. This is the fourth input argument. `[t,y] = solver(odefun,tspan,y0,options)` You can create options using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for the solvers.

`T`	Column vector of time points.
`Y`	Solution array. Each row in `Y` corresponds to the solution at a time returned in the corresponding row of `T`.
`TE`	The time at which an event occurs.
`YE`	The solution at the time of the event.

`IE`	The index `i` of the event function that vanishes.
`sol`	Structure to evaluate the solution.

Description

`[T,Y] = solver(odefun,tspan,y0)` with `tspan = [t0 tf]` integrates the system of differential equations $y' = f(t,y)$ from time `t0` to `tf` with initial conditions `y0`. The first input argument, `odefun`, is a function handle. The function, `f = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `f` corresponding to $f(t,y)$. Each row in the solution array `Y` corresponds to a time returned in column vector `T`. To obtain solutions at the specific times `t0`, `t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`[T,Y] = solver(odefun,tspan,y0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used properties include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components are `1e-6` by default). If certain components of the solution must be nonnegative, use the `odeset` function to set the `NonNegative` property to the indices of these components. See `odeset` for details.

`[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)` solves as above while also finding where functions of (t,y), called event functions, are zero. For each event function, you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, e.g., `events` or `@events`, and creating a function `[value,isterminal,direction] = events(t,y)`. For the `i`th event function in `events`,

- `value(i)` is the value of the function.
- `isterminal(i) = 1`, if the integration is to terminate at a zero of this event function and `0` otherwise.

- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Corresponding entries in `TE`, `YE`, and `IE` return, respectively, the time at which an event occurs, the solution at the time of the event, and the index `i` of the event function that vanishes.

`sol = solver(odefun,[t0 tf],y0...)` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0,tf]`. You must pass `odefun` as a function handle. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver.
`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.
`sol.solver`	Solver name.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen.

By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

For the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, the Jacobian matrix $\partial f/\partial y$ is critical to reliability and efficiency. Use `odeset` to set `Jacobian` to `@FJAC` if `FJAC(T,Y)` returns the Jacobian $\partial f/\partial y$ or to the matrix $\partial f/\partial y$ if the Jacobian is constant. If the `Jacobian` property is not set (the default), $\partial f/\partial y$ is approximated by finite differences. Set the `Vectorized` property 'on' if the ODE function is coded so that `odefun(T,[Y1,Y2 ...])` returns `[odefun(T,Y1),odefun(T,Y2) ...]`. If $\partial f/\partial y$ is a sparse matrix, set the `JPattern` property to the sparsity pattern of $\partial f/\partial y$, i.e., a sparse matrix `S` with `S(i,j) = 1` if the `i`th component of $f(t,y)$ depends on the `j`th component of y, and 0 otherwise.

The solvers of the ODE suite can solve problems of the form $M(t,y)y' = f(t,y)$, with time- and state-dependent mass matrix M. (The `ode23s` solver can solve only equations with constant mass matrices.) If a problem has a mass matrix, create a function `M = MASS(t,y)` that returns the value of the mass matrix, and use `odeset` to set the `Mass` property to `@MASS`. If the mass matrix is constant, the matrix should be used as the value of the `Mass` property. Problems with state-dependent mass matrices are more difficult:

- If the mass matrix does not depend on the state variable y and the function `MASS` is to be called with one input argument, `t`, set the `MStateDependence` property to '`none`'.
- If the mass matrix depends weakly on y, set `MStateDependence` to '`weak`' (the default); otherwise, set it to '`strong`'. In either case, the function `MASS` is called with the two arguments `(t,y)`.

If there are many differential equations, it is important to exploit sparsity:

- Return a sparse $M(t,y)$.

- Supply the sparsity pattern of $\partial f/\partial y$ using the `JPattern` property or a sparse $\partial f/\partial y$ using the `Jacobian` property.
- For strongly state-dependent $M(t,y)$, set `MvPattern` to a sparse matrix `S` with `S(i,j) = 1` if for any `k`, the `(i,k)` component of $M(t,y)$ depends on component `j` of y, and `0` otherwise.

If the mass matrix M is singular, then $M(t,y)y' = f(t,y)$ is a system of differential algebraic equations. DAEs have solutions only when y_0 is consistent, that is, if there is a vector yp_0 such that $M(t_0,y_0)yp_0 = f(t_0,y_0)$. The `ode15s` and `ode23t` solvers can solve DAEs of index 1 provided that `y0` is sufficiently close to being consistent. If there is a mass matrix, you can use `odeset` to set the `MassSingular` property to `'yes'`, `'no'`, or `'maybe'`. The default value of `'maybe'` causes the solver to test whether the problem is a DAE. You can provide `yp0` as the value of the `InitialSlope` property. The default is the zero vector. If a problem is a DAE, and `y0` and `yp0` are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. When solving DAEs, it is very advantageous to formulate the problem so that M is a diagonal matrix (a semi-explicit DAE).

Solver	Problem Type	Order of Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.
`ode23`	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.

Solver	Problem Type	Order of Accuracy	When to Use
`ode113`	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.
`ode15s`	Stiff	Low to medium	If `ode45` is slow because the problem is stiff.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
`ode23t`	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
`ode23tb`	Stiff	Low	If using crude error tolerances to solve stiff systems.

The algorithms used in the ODE solvers vary according to order of accuracy [6] and the type of systems (stiff or nonstiff) they are designed to solve. See “Algorithms” on page 1-3670 for more details.

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and “Integrator Options” in the MATLAB Mathematics documentation.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
RelTol, AbsTol, NormControl	√	√	√	√	√	√	√
OutputFcn, OutputSel, Refine, Stats	√	√	√	√	√	√	√
NonNegative	√	√	√	√ *	—	√ *	√ *
Events	√	√	√	√	√	√	√
MaxStep, InitialStep	√	√	√	√	√	√	√
Jacobian, JPattern, Vectorized	—	—	—	√	√	√	√
Mass MStateDependence MvPattern MassSingular	√ √ — —	√ √ — —	√ √ — —	√ √ √ √	√ — — —	√ √ √ √	√ √ √ —
InitialSlope	—	—	—	√	—	√	—
MaxOrder, BDF	—	—	—	√	—	—	—

Note You can use the `NonNegative` parameter with `ode15s`, `ode23t`, and `ode23tb` only for those problems for which there is no mass matrix.

Examples

Example 1

An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces.

$$\begin{aligned} y_1' &= y_2 y_3 & y_1(0) &= 0 \\ y_2' &= -y_1 y_3 & y_2(0) &= 1 \\ y_3' &= -0.51 y_1 y_2 & y_3(0) &= 1 \end{aligned}$$

To simulate this system, create a function `rigid` containing the equations

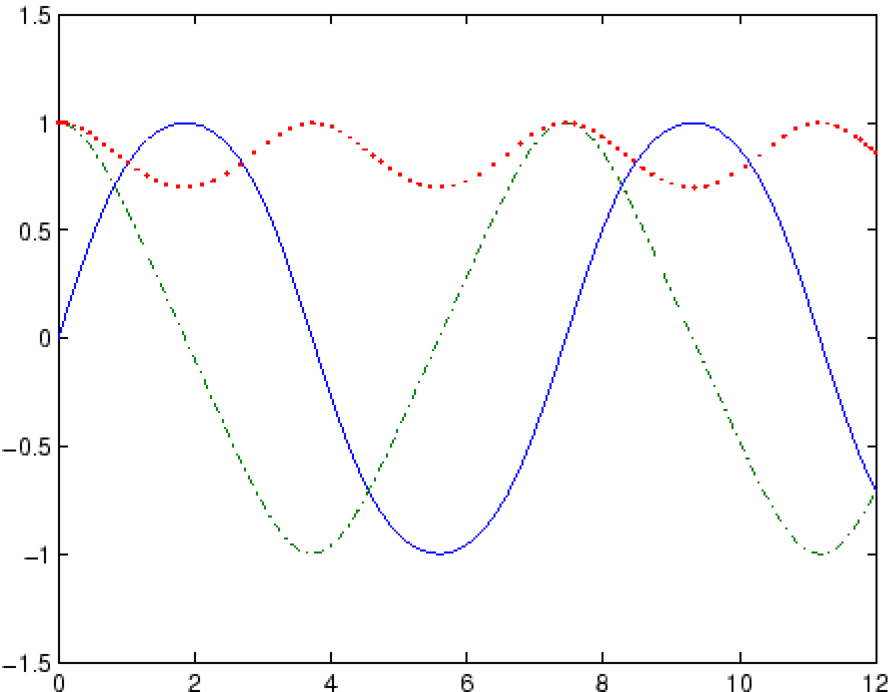
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time `0`.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[T,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

Plotting the columns of the returned array `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'.')
```



Example 2

An example of a stiff system is provided by the van der Pol equations in relaxation oscillation. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$\begin{aligned} y_1' &= y_2 & \quad y_1(0) &= 2 \\ y_2' &= 1000(1 - y_1^2)y_2 - y_1 & \quad y_2(0) &= 0 \end{aligned}$$

To simulate this system, create a function `vdp1000` containing the equations

```
function dy = vdp1000(t,y)
```

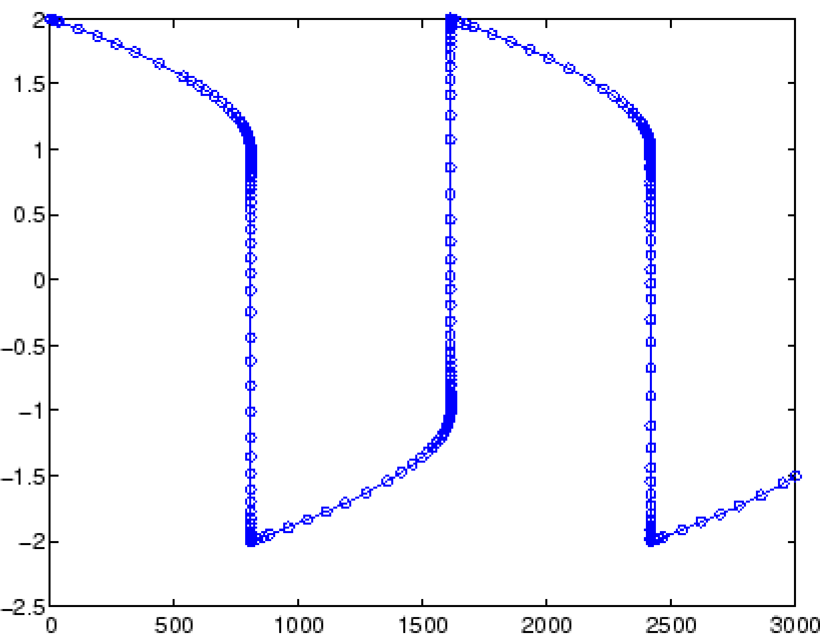
```
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

For this problem, we will use the default relative and absolute tolerances (`1e-3` and `1e-6`, respectively) and solve on a time interval of `[0 3000]` with initial condition vector `[2 0]` at time `0`.

```
[T,Y] = ode15s(@vdp1000,[0 3000],[2 0]);
```

Plotting the first column of the returned matrix `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-o')
```



Example 3

This example solves an ordinary differential equation with time-dependent terms.

Consider the following ODE, with time-dependent parameters defined only through the set of data points given in two vectors:

```
y'(t) + f(t)y(t) = g(t)
```

The initial condition is `y(0) = 0`, where the function `f(t)` is defined through the `n`-by-`1` vectors `tf` and `f`, and the function `g(t)` is defined through the `m`-by-`1` vectors `tg` and `g`.

First, define the time-dependent parameters `f(t)` and `g(t)` as the following:

```
ft = linspace(0,5,25); % Generate t for f
f = ft.^2 - ft - 3; % Generate f(t)
gt = linspace(1,6,25); % Generate t for g
g = 3*sin(gt-0.25); % Generate g(t)
```

Write a function to interpolate the data sets specified above to obtain the value of the time-dependent terms at the specified time:

```
function dydt = myode(t,y,ft,f,gt,g)
f = interp1(ft,f,t); % Interpolate the data set (ft,f) at time t
g = interp1(gt,g,t); % Interpolate the data set (gt,g) at time t
dydt = -f.*y + g; % Evalute ODE at time t
```

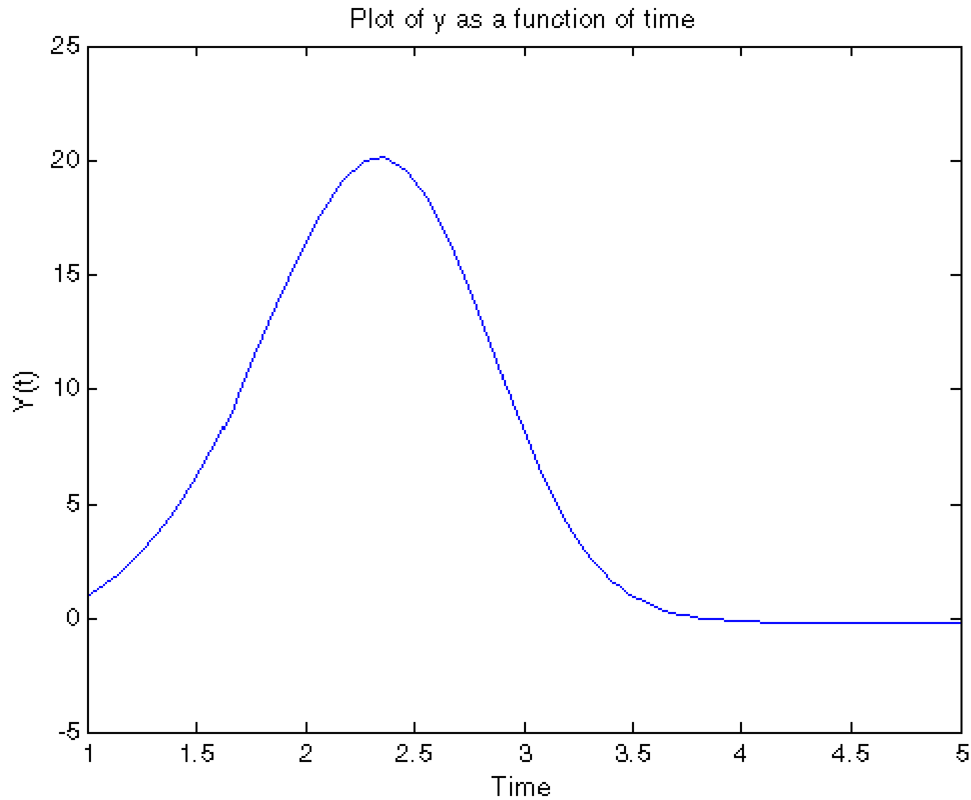
Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
Tspan = [1 5]; % Solve from t=1 to t=5
IC = 1; % y(t=0) = 1
[T Y] = ode45(@(t,y) myode(t,y,ft,f,gt,g),Tspan,IC); % Solve ODE
```

Plot the solution `y(t)` as a function of time:

```
plot(T, Y);
```

```
title('Plot of y as a function of time');
xlabel('Time'); ylabel('Y(t)');
```



Algorithms

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing `y(t`$_n$`)`, it needs only the solution at the immediately preceding time point, `y(t`$_{n-1}$`)`. In general, `ode45` is the best function to apply as a *first try* for most problems. [3]

`ode23` is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude

tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver. [2]

`ode113` is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE file function is particularly expensive to evaluate. `ode113` is a *multistep* solver — it normally needs the solutions at several preceding time points to compute the current solution. [7]

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` when `ode45` fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem. [9], [10]

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective. [9]

`ode23t` is an implementation of the trapezoidal rule using a "free" interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve DAEs. [10]

`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances. [8], [1]

References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R. Smith, "Transient Simulation of Silicon Devices and Circuits," *IEEE Trans. CAD*, 4 (1985), pp. 436–451.

[2] Bogacki, P. and L. F. Shampine, "A 3(2) pair of Runge-Kutta formulas," *Appl. Math. Letters*, Vol. 2, 1989, pp. 321–325.

[3] Dormand, J. R. and P. J. Prince, "A family of embedded Runge-Kutta formulae," *J. Comp. Appl. Math.*, Vol. 6, 1980, pp. 19–26.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, San Francisco, 1975.

[8] Shampine, L. F. and M. E. Hosea, "Analysis and Implementation of TR-BDF2," *Applied Numerical Mathematics 20*, 1996.

[9] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing,* Vol. 18, 1997, pp. 1–22.

[10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, "Solving Index-1 DAEs in MATLAB and Simulink," *SIAM Review*, Vol. 41, 1999, pp. 538–552.

See Also

`deval` | `ode15i` | `odeget` | `odeset` | `function_handle`

Purpose

Solve stiff differential equations; low order method

Syntax

```
[T,Y] = solver(odefun,tspan,y0)
[T,Y] = solver(odefun,tspan,y0,options)
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
sol = solver(odefun,[t0 tf],y0...)
```

This page contains an overview of the solver functions: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. You can call any of these solvers by substituting the placeholder, *`solver`*, with any of the function names.

Arguments

The following table describes the input arguments to the solvers.

`odefun`	A function handle that evaluates the right side of the differential equations. All solvers solve systems of equations in the form $y' = f(t,y)$ or problems that involve a mass matrix, $M(t,y)y' = f(t,y)$. The `ode23s` solver can solve only equations with constant mass matrices. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`. For `tspan` vectors with two elements `[t0 tf]`, the solver returns the solution evaluated at every integration step. For `tspan` vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.

	Specifying `tspan` with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from `tspan(1)` to `tspan(end)`. All solvers in the ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in `tspan`, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points. Specifying `tspan` with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management.
`y0`	A vector of initial conditions.
`options`	Structure of optional parameters that change the default integration properties. This is the fourth input argument. `[t,y] = solver(odefun,tspan,y0,options)` You can create options using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for the solvers.

`T`	Column vector of time points.
`Y`	Solution array. Each row in `Y` corresponds to the solution at a time returned in the corresponding row of `T`.
`TE`	The time at which an event occurs.
`YE`	The solution at the time of the event.

`IE`	The index `i` of the event function that vanishes.
`sol`	Structure to evaluate the solution.

Description

`[T,Y] = solver(odefun,tspan,y0)` with `tspan = [t0 tf]` integrates the system of differential equations $y' = f(t,y)$ from time `t0` to `tf` with initial conditions `y0`. The first input argument, `odefun`, is a function handle. The function, `f = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `f` corresponding to $f(t,y)$. Each row in the solution array `Y` corresponds to a time returned in column vector `T`. To obtain solutions at the specific times `t0`, `t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`[T,Y] = solver(odefun,tspan,y0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used properties include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components are `1e-6` by default). If certain components of the solution must be nonnegative, use the `odeset` function to set the `NonNegative` property to the indices of these components. See `odeset` for details.

`[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)` solves as above while also finding where functions of (t,y), called event functions, are zero. For each event function, you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, e.g., `events` or `@events`, and creating a function `[value,isterminal,direction] = events(t,y)`. For the `i`th event function in `events`,

- `value(i)` is the value of the function.
- `isterminal(i) = 1`, if the integration is to terminate at a zero of this event function and `0` otherwise.

- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Corresponding entries in `TE`, `YE`, and `IE` return, respectively, the time at which an event occurs, the solution at the time of the event, and the index `i` of the event function that vanishes.

`sol = `*`solver`*`(odefun,[t0 tf],y0...)` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0,tf]`. You must pass `odefun` as a function handle. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver.
`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.
`sol.solver`	Solver name.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen.

By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

For the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, the Jacobian matrix $\partial f/\partial y$ is critical to reliability and efficiency. Use `odeset` to set `Jacobian` to `@FJAC` if `FJAC(T,Y)` returns the Jacobian $\partial f/\partial y$ or to the matrix $\partial f/\partial y$ if the Jacobian is constant. If the `Jacobian` property is not set (the default), $\partial f/\partial y$ is approximated by finite differences. Set the `Vectorized` property '`on`' if the ODE function is coded so that `odefun(T,[Y1,Y2 ...])` returns `[odefun(T,Y1),odefun(T,Y2) ...]`. If $\partial f/\partial y$ is a sparse matrix, set the `JPattern` property to the sparsity pattern of $\partial f/\partial y$, i.e., a sparse matrix `S` with `S(i,j) = 1` if the `i`th component of $f(t,y)$ depends on the `j`th component of y, and 0 otherwise.

The solvers of the ODE suite can solve problems of the form $M(t,y)y' = f(t,y)$, with time- and state-dependent mass matrix M. (The `ode23s` solver can solve only equations with constant mass matrices.) If a problem has a mass matrix, create a function `M = MASS(t,y)` that returns the value of the mass matrix, and use `odeset` to set the `Mass` property to `@MASS`. If the mass matrix is constant, the matrix should be used as the value of the `Mass` property. Problems with state-dependent mass matrices are more difficult:

- If the mass matrix does not depend on the state variable y and the function `MASS` is to be called with one input argument, `t`, set the `MStateDependence` property to '`none`'.
- If the mass matrix depends weakly on y, set `MStateDependence` to '`weak`' (the default); otherwise, set it to '`strong`'. In either case, the function `MASS` is called with the two arguments `(t,y)`.

If there are many differential equations, it is important to exploit sparsity:

- Return a sparse $M(t,y)$.

- Supply the sparsity pattern of $\partial f/\partial y$ using the `JPattern` property or a sparse $\partial f/\partial y$ using the `Jacobian` property.
- For strongly state-dependent $M(t,y)$, set `MvPattern` to a sparse matrix `S` with `S(i,j) = 1` if for any `k`, the `(i,k)` component of $M(t,y)$ depends on component `j` of y, and `0` otherwise.

If the mass matrix M is singular, then $M(t,y)y' = f(t,y)$ is a system of differential algebraic equations. DAEs have solutions only when y_0 is consistent, that is, if there is a vector yp_0 such that $M(t_0,y_0)yp_0 = f(t_0,y_0)$. The `ode15s` and `ode23t` solvers can solve DAEs of index 1 provided that `y0` is sufficiently close to being consistent. If there is a mass matrix, you can use `odeset` to set the `MassSingular` property to `'yes'`, `'no'`, or `'maybe'`. The default value of `'maybe'` causes the solver to test whether the problem is a DAE. You can provide `yp0` as the value of the `InitialSlope` property. The default is the zero vector. If a problem is a DAE, and `y0` and `yp0` are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. When solving DAEs, it is very advantageous to formulate the problem so that M is a diagonal matrix (a semi-explicit DAE).

Solver	Problem Type	Order of Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.
`ode23`	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.

Solver	Problem Type	Order of Accuracy	When to Use
`ode113`	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.
`ode15s`	Stiff	Low to medium	If `ode45` is slow because the problem is stiff.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
`ode23t`	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
`ode23tb`	Stiff	Low	If using crude error tolerances to solve stiff systems.

The algorithms used in the ODE solvers vary according to order of accuracy [6] and the type of systems (stiff or nonstiff) they are designed to solve. See "Algorithms" on page 1-3685 for more details.

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and "Integrator Options" in the MATLAB Mathematics documentation.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
RelTol, AbsTol, NormControl	√	√	√	√	√	√	√
OutputFcn, OutputSel, Refine, Stats	√	√	√	√	√	√	√
NonNegative	√	√	√	√ *	—	√ *	√ *
Events	√	√	√	√	√	√	√
MaxStep, InitialStep	√	√	√	√	√	√	√
Jacobian, JPattern, Vectorized	—	—	—	√	√	√	√
Mass MStateDependence MvPattern MassSingular	√ √ — —	√ √ — —	√ √ — —	√ √ √ √	√ — — —	√ √ √ √	√ √ √ —
InitialSlope	—	—	—	√	—	√	—
MaxOrder, BDF	—	—	—	√	—	—	—

Note You can use the `NonNegative` parameter with `ode15s`, `ode23t`, and `ode23tb` only for those problems for which there is no mass matrix.

Examples

Example 1

An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces.

$$
\begin{aligned}
y_1' &= y_2 y_3 & y_1(0) &= 0 \\
y_2' &= -y_1 y_3 & y_2(0) &= 1 \\
y_3' &= -0.51 y_1 y_2 & y_3(0) &= 1
\end{aligned}
$$

To simulate this system, create a function `rigid` containing the equations

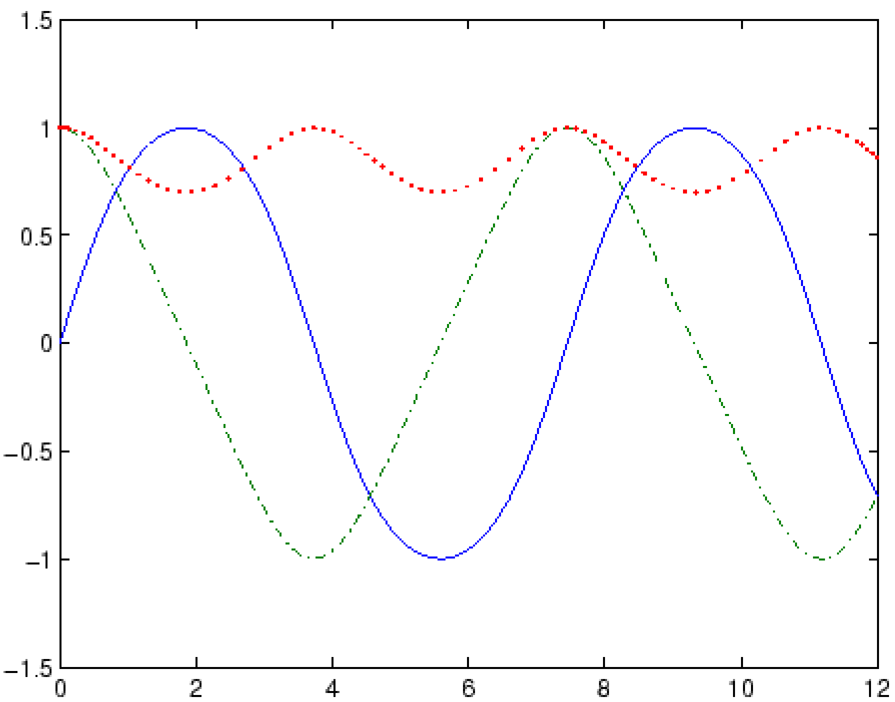
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time `0`.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[T,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

Plotting the columns of the returned array `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'.')
```



Example 2

An example of a stiff system is provided by the van der Pol equations in relaxation oscillation. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$
\begin{aligned}
y_1' &= y_2 & y_1(0) &= 2 \\
y_2' &= 1000(1 - y_1^2)y_2 - y_1 & y_2(0) &= 0
\end{aligned}
$$

To simulate this system, create a function `vdp1000` containing the equations

```
function dy = vdp1000(t,y)
```

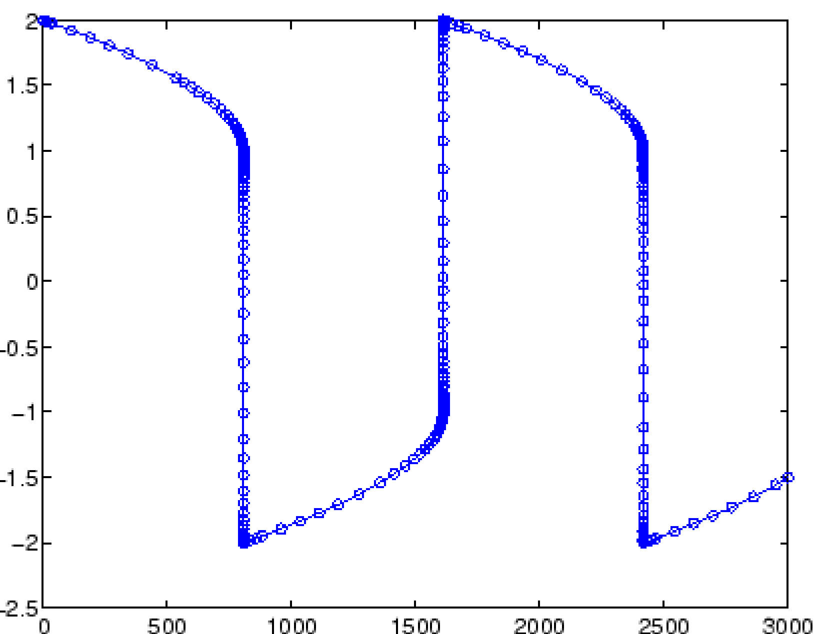
```
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

For this problem, we will use the default relative and absolute tolerances (`1e-3` and `1e-6`, respectively) and solve on a time interval of `[0 3000]` with initial condition vector `[2 0]` at time `0`.

```
[T,Y] = ode15s(@vdp1000,[0 3000],[2 0]);
```

Plotting the first column of the returned matrix `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-o')
```



Example 3

This example solves an ordinary differential equation with time-dependent terms.

Consider the following ODE, with time-dependent parameters defined only through the set of data points given in two vectors:

```
y'(t) + f(t)y(t) = g(t)
```

The initial condition is `y(0) = 0`, where the function `f(t)` is defined through the `n`-by-`1` vectors `tf` and `f`, and the function `g(t)` is defined through the `m`-by-`1` vectors `tg` and `g`.

First, define the time-dependent parameters `f(t)` and `g(t)` as the following:

```
ft = linspace(0,5,25); % Generate t for f
f = ft.^2 - ft - 3; % Generate f(t)
gt = linspace(1,6,25); % Generate t for g
g = 3*sin(gt-0.25); % Generate g(t)
```

Write a function to interpolate the data sets specified above to obtain the value of the time-dependent terms at the specified time:

```
function dydt = myode(t,y,ft,f,gt,g)
f = interp1(ft,f,t); % Interpolate the data set (ft,f) at time t
g = interp1(gt,g,t); % Interpolate the data set (gt,g) at time t
dydt = -f.*y + g; % Evalute ODE at time t
```

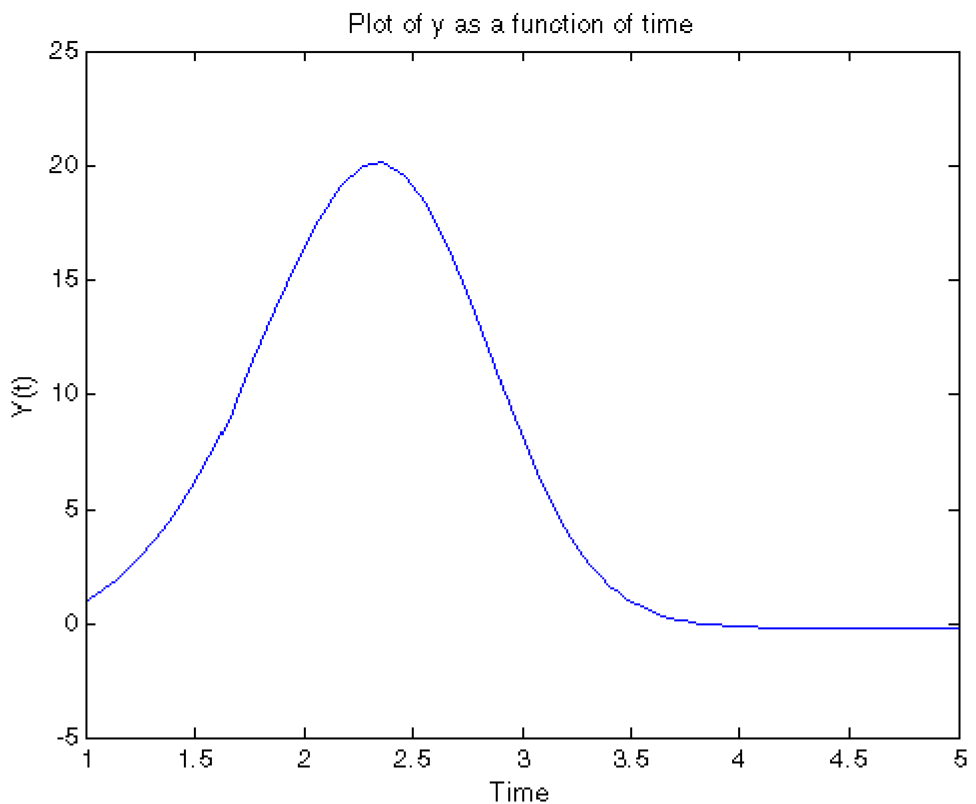
Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
Tspan = [1 5]; % Solve from t=1 to t=5
IC = 1; % y(t=0) = 1
[T Y] = ode45(@(t,y) myode(t,y,ft,f,gt,g),Tspan,IC); % Solve ODE
```

Plot the solution `y(t)` as a function of time:

```
plot(T, Y);
```

```
title('Plot of y as a function of time');
xlabel('Time'); ylabel('Y(t)');
```



Algorithms

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing `y(t`$_n$`)`, it needs only the solution at the immediately preceding time point, `y(t`$_{n-1}$`)`. In general, `ode45` is the best function to apply as a *first try* for most problems. [3]

`ode23` is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude

tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver. [2]

`ode113` is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE file function is particularly expensive to evaluate. `ode113` is a *multistep* solver — it normally needs the solutions at several preceding time points to compute the current solution. [7]

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` when `ode45` fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem. [9], [10]

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective. [9]

`ode23t` is an implementation of the trapezoidal rule using a "free" interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve DAEs. [10]

`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances. [8], [1]

References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R. Smith, "Transient Simulation of Silicon Devices and Circuits," *IEEE Trans. CAD*, 4 (1985), pp. 436–451.

[2] Bogacki, P. and L. F. Shampine, "A 3(2) pair of Runge-Kutta formulas," *Appl. Math. Letters*, Vol. 2, 1989, pp. 321–325.

[3] Dormand, J. R. and P. J. Prince, "A family of embedded Runge-Kutta formulae," *J. Comp. Appl. Math.*, Vol. 6, 1980, pp. 19–26.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, San Francisco, 1975.

[8] Shampine, L. F. and M. E. Hosea, "Analysis and Implementation of TR-BDF2," *Applied Numerical Mathematics 20*, 1996.

[9] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing,* Vol. 18, 1997, pp. 1–22.

[10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, "Solving Index-1 DAEs in MATLAB and Simulink," *SIAM Review*, Vol. 41, 1999, pp. 538–552.

See Also

`deval` | `ode15i` | `odeget` | `odeset` | `function_handle`

Purpose

Solve moderately stiff ODEs and DAEs; trapezoidal rule

Syntax

```
[T,Y] = solver(odefun,tspan,y0)
[T,Y] = solver(odefun,tspan,y0,options)
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
sol = solver(odefun,[t0 tf],y0...)
```

This page contains an overview of the solver functions: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. You can call any of these solvers by substituting the placeholder, *`solver`*, with any of the function names.

Arguments

The following table describes the input arguments to the solvers.

`odefun`	A function handle that evaluates the right side of the differential equations. All solvers solve systems of equations in the form $y' = f(t,y)$ or problems that involve a mass matrix, $M(t,y)y' = f(t,y)$. The `ode23s` solver can solve only equations with constant mass matrices. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`. For `tspan` vectors with two elements `[t0 tf]`, the solver returns the solution evaluated at every integration step. For `tspan` vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.

	Specifying `tspan` with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from `tspan(1)` to `tspan(end)`. All solvers in the ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in `tspan`, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points. Specifying `tspan` with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management.
`y0`	A vector of initial conditions.
`options`	Structure of optional parameters that change the default integration properties. This is the fourth input argument. `[t,y] = solver(odefun,tspan,y0,options)` You can create options using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for the solvers.

`T`	Column vector of time points.
`Y`	Solution array. Each row in `Y` corresponds to the solution at a time returned in the corresponding row of `T`.
`TE`	The time at which an event occurs.
`YE`	The solution at the time of the event.

IE	The index i of the event function that vanishes.
sol	Structure to evaluate the solution.

Description

[T,Y] = *solver*(odefun,tspan,y0) with tspan = [t0 tf] integrates the system of differential equations $y' = f(t,y)$ from time t0 to tf with initial conditions y0. The first input argument, odefun, is a function handle. The function, f = odefun(t,y), for a scalar t and a column vector y, must return a column vector f corresponding to $f(t,y)$. Each row in the solution array Y corresponds to a time returned in column vector T. To obtain solutions at the specific times t0, t1,...,tf (all increasing or all decreasing), use tspan = [t0,t1,...,tf].

"Parameterizing Functions" explains how to provide additional parameters to the function fun, if necessary.

[T,Y] = *solver*(odefun,tspan,y0,options) solves as above with default integration parameters replaced by property values specified in options, an argument created with the odeset function. Commonly used properties include a scalar relative error tolerance RelTol (1e-3 by default) and a vector of absolute error tolerances AbsTol (all components are 1e-6 by default). If certain components of the solution must be nonnegative, use the odeset function to set the NonNegative property to the indices of these components. See odeset for details.

[T,Y,TE,YE,IE] = *solver*(odefun,tspan,y0,options) solves as above while also finding where functions of (t,y), called event functions, are zero. For each event function, you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the 'Events' property to a function, e.g., events or @events, and creating a function [value,isterminal,direction] = events(t,y). For the ith event function in events,

- value(i) is the value of the function.
- isterminal(i) = 1, if the integration is to terminate at a zero of this event function and 0 otherwise.

- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Corresponding entries in `TE`, `YE`, and `IE` return, respectively, the time at which an event occurs, the solution at the time of the event, and the index `i` of the event function that vanishes.

`sol = solver(odefun,[t0 tf],y0...)` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0,tf]`. You must pass `odefun` as a function handle. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver.
`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.
`sol.solver`	Solver name.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen.

By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

For the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, the Jacobian matrix $\partial f/\partial y$ is critical to reliability and efficiency. Use `odeset` to set `Jacobian` to `@FJAC` if `FJAC(T,Y)` returns the Jacobian $\partial f/\partial y$ or to the matrix $\partial f/\partial y$ if the Jacobian is constant. If the `Jacobian` property is not set (the default), $\partial f/\partial y$ is approximated by finite differences. Set the `Vectorized` property '`on`' if the ODE function is coded so that `odefun(T,[Y1,Y2 ...])` returns `[odefun(T,Y1),odefun(T,Y2) ...]`. If $\partial f/\partial y$ is a sparse matrix, set the `JPattern` property to the sparsity pattern of $\partial f/\partial y$, i.e., a sparse matrix `S` with `S(i,j) = 1` if the `i`th component of $f(t,y)$ depends on the `j`th component of y, and 0 otherwise.

The solvers of the ODE suite can solve problems of the form $M(t,y)y' = f(t,y)$, with time- and state-dependent mass matrix M. (The `ode23s` solver can solve only equations with constant mass matrices.) If a problem has a mass matrix, create a function `M = MASS(t,y)` that returns the value of the mass matrix, and use `odeset` to set the `Mass` property to `@MASS`. If the mass matrix is constant, the matrix should be used as the value of the `Mass` property. Problems with state-dependent mass matrices are more difficult:

- If the mass matrix does not depend on the state variable y and the function `MASS` is to be called with one input argument, `t`, set the `MStateDependence` property to '`none`'.
- If the mass matrix depends weakly on y, set `MStateDependence` to '`weak`' (the default); otherwise, set it to '`strong`'. In either case, the function `MASS` is called with the two arguments `(t,y)`.

If there are many differential equations, it is important to exploit sparsity:

- Return a sparse $M(t,y)$.

- Supply the sparsity pattern of $\partial f/\partial y$ using the `JPattern` property or a sparse $\partial f/\partial y$ using the `Jacobian` property.
- For strongly state-dependent $M(t,y)$, set `MvPattern` to a sparse matrix `S` with `S(i,j) = 1` if for any `k`, the `(i,k)` component of $M(t,y)$ depends on component `j` of y, and `0` otherwise.

If the mass matrix M is singular, then $M(t,y)y' = f(t,y)$ is a system of differential algebraic equations. DAEs have solutions only when y_0 is consistent, that is, if there is a vector yp_0 such that $M(t_0,y_0)yp_0 = f(t_0,y_0)$. The `ode15s` and `ode23t` solvers can solve DAEs of index 1 provided that `y0` is sufficiently close to being consistent. If there is a mass matrix, you can use `odeset` to set the `MassSingular` property to `'yes'`, `'no'`, or `'maybe'`. The default value of `'maybe'` causes the solver to test whether the problem is a DAE. You can provide `yp0` as the value of the `InitialSlope` property. The default is the zero vector. If a problem is a DAE, and `y0` and `yp0` are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. When solving DAEs, it is very advantageous to formulate the problem so that M is a diagonal matrix (a semi-explicit DAE).

Solver	Problem Type	Order of Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.
`ode23`	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.

Solver	Problem Type	Order of Accuracy	When to Use
`ode113`	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.
`ode15s`	Stiff	Low to medium	If `ode45` is slow because the problem is stiff.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
`ode23t`	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
`ode23tb`	Stiff	Low	If using crude error tolerances to solve stiff systems.

The algorithms used in the ODE solvers vary according to order of accuracy [6] and the type of systems (stiff or nonstiff) they are designed to solve. See “Algorithms” on page 1-3700 for more details.

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and “Integrator Options” in the MATLAB Mathematics documentation.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
RelTol, AbsTol, NormControl	√	√	√	√	√	√	√
OutputFcn, OutputSel, Refine, Stats	√	√	√	√	√	√	√
NonNegative	√	√	√	√ *	—	√ *	√ *
Events	√	√	√	√	√	√	√
MaxStep, InitialStep	√	√	√	√	√	√	√
Jacobian, JPattern, Vectorized	—	—	—	√	√	√	√
Mass MStateDependence MvPattern MassSingular	√ √ — —	√ √ — —	√ √ — —	√ √ √ √	√ — — —	√ √ √ √	√ √ √ —
InitialSlope	—	—	—	√	—	√	—
MaxOrder, BDF	—	—	—	√	—	—	—

Note You can use the `NonNegative` parameter with `ode15s`, `ode23t`, and `ode23tb` only for those problems for which there is no mass matrix.

Examples

Example 1

An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces.

$$
\begin{aligned}
y_1' &= y_2 y_3 & y_1(0) &= 0 \\
y_2' &= -y_1 y_3 & y_2(0) &= 1 \\
y_3' &= -0.51 y_1 y_2 & y_3(0) &= 1
\end{aligned}
$$

To simulate this system, create a function `rigid` containing the equations

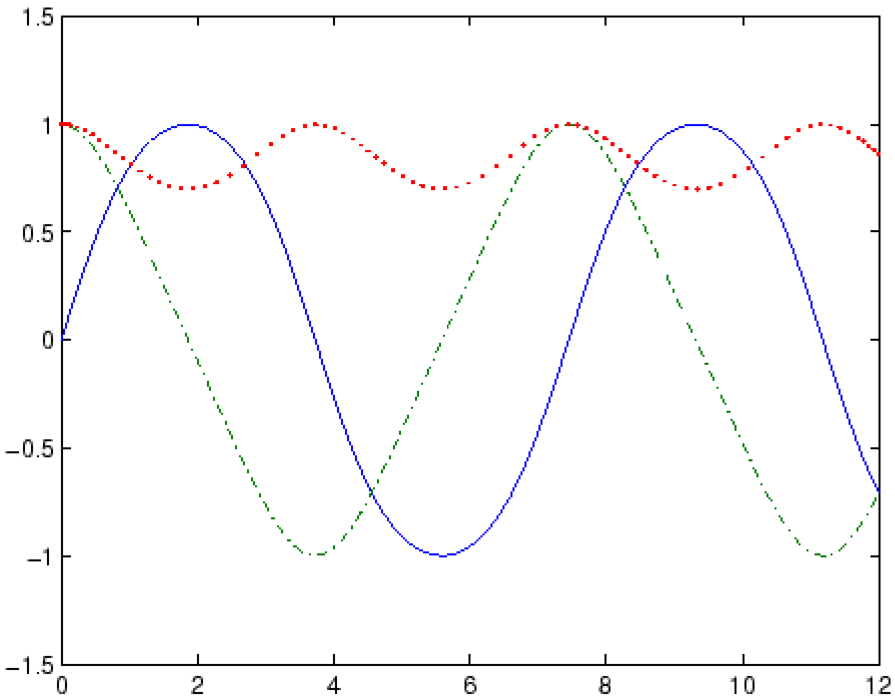
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time `0`.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[T,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

Plotting the columns of the returned array `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'.')
```



Example 2

An example of a stiff system is provided by the van der Pol equations in relaxation oscillation. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$y'_1 = y_2 \qquad y_1(0) = 2$$
$$y'_2 = 1000(1 - y_1^2)y_2 - y_1 \qquad y_2(0) = 0$$

To simulate this system, create a function `vdp1000` containing the equations

```
function dy = vdp1000(t,y)
```

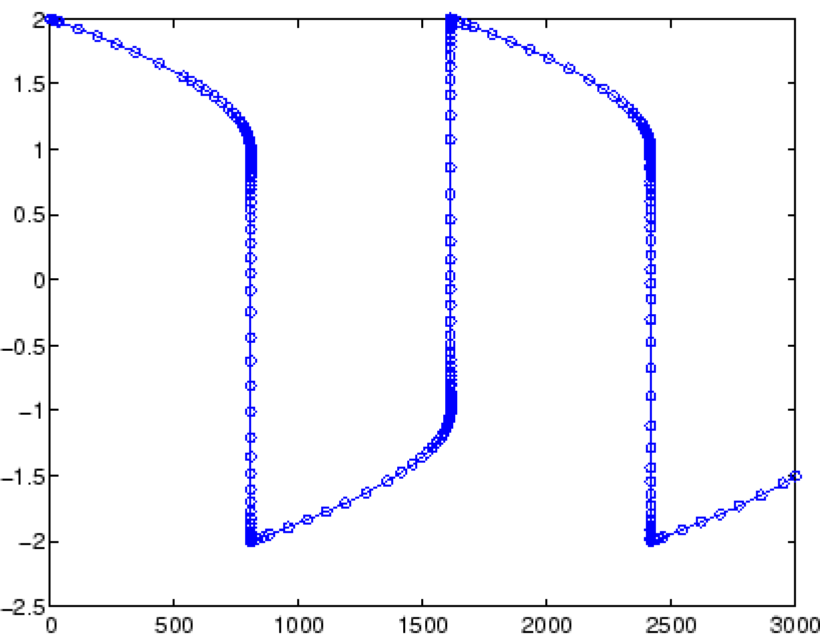
```
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

For this problem, we will use the default relative and absolute tolerances (`1e-3` and `1e-6`, respectively) and solve on a time interval of `[0 3000]` with initial condition vector `[2 0]` at time `0`.

```
[T,Y] = ode15s(@vdp1000,[0 3000],[2 0]);
```

Plotting the first column of the returned matrix `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-o')
```



Example 3

This example solves an ordinary differential equation with time-dependent terms.

Consider the following ODE, with time-dependent parameters defined only through the set of data points given in two vectors:

```
y'(t) + f(t)y(t) = g(t)
```

The initial condition is `y(0) = 0`, where the function `f(t)` is defined through the `n`-by-`1` vectors `tf` and `f`, and the function `g(t)` is defined through the `m`-by-`1` vectors `tg` and `g`.

First, define the time-dependent parameters `f(t)` and `g(t)` as the following:

```
ft = linspace(0,5,25); % Generate t for f
f = ft.^2 - ft - 3; % Generate f(t)
gt = linspace(1,6,25); % Generate t for g
g = 3*sin(gt-0.25); % Generate g(t)
```

Write a function to interpolate the data sets specified above to obtain the value of the time-dependent terms at the specified time:

```
function dydt = myode(t,y,ft,f,gt,g)
f = interp1(ft,f,t); % Interpolate the data set (ft,f) at time t
g = interp1(gt,g,t); % Interpolate the data set (gt,g) at time t
dydt = -f.*y + g; % Evalute ODE at time t
```

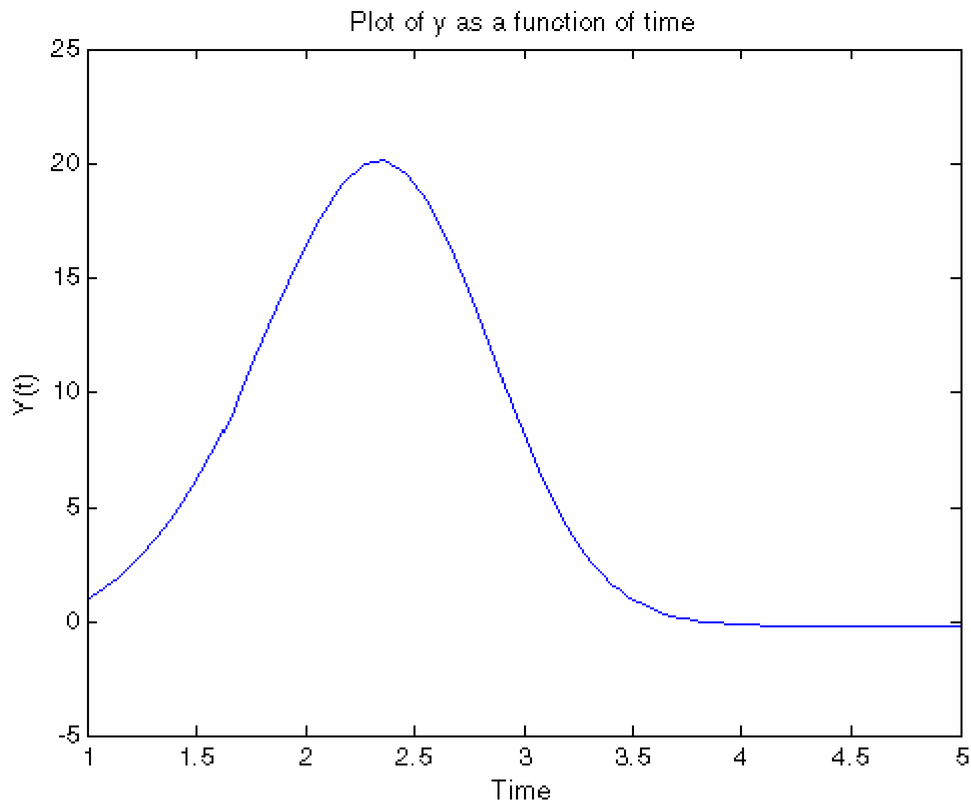
Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
Tspan = [1 5]; % Solve from t=1 to t=5
IC = 1; % y(t=0) = 1
[T Y] = ode45(@(t,y) myode(t,y,ft,f,gt,g),Tspan,IC); % Solve ODE
```

Plot the solution `y(t)` as a function of time:

```
plot(T, Y);
```

```
title('Plot of y as a function of time');
xlabel('Time'); ylabel('Y(t)');
```



Algorithms

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing `y(`t_n`)`, it needs only the solution at the immediately preceding time point, `y(`t_{n-1}`)`. In general, `ode45` is the best function to apply as a *first try* for most problems. [3]

`ode23` is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude

tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver. [2]

`ode113` is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE file function is particularly expensive to evaluate. `ode113` is a *multistep* solver — it normally needs the solutions at several preceding time points to compute the current solution. [7]

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` when `ode45` fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem. [9], [10]

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective. [9]

`ode23t` is an implementation of the trapezoidal rule using a "free" interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve DAEs. [10]

`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances. [8], [1]

References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R. Smith, "Transient Simulation of Silicon Devices and Circuits," *IEEE Trans. CAD*, 4 (1985), pp. 436–451.

[2] Bogacki, P. and L. F. Shampine, “A 3(2) pair of Runge-Kutta formulas,” *Appl. Math. Letters*, Vol. 2, 1989, pp. 321–325.

[3] Dormand, J. R. and P. J. Prince, “A family of embedded Runge-Kutta formulae,” *J. Comp. Appl. Math.*, Vol. 6, 1980, pp. 19–26.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, San Francisco, 1975.

[8] Shampine, L. F. and M. E. Hosea, “Analysis and Implementation of TR-BDF2,” *Applied Numerical Mathematics 20*, 1996.

[9] Shampine, L. F. and M. W. Reichelt, “The MATLAB ODE Suite,” *SIAM Journal on Scientific Computing,* Vol. 18, 1997, pp. 1–22.

[10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, “Solving Index-1 DAEs in MATLAB and Simulink,” *SIAM Review*, Vol. 41, 1999, pp. 538–552.

See Also

`deval` | `ode15i` | `odeget` | `odeset` | `function_handle`

Purpose

Solve stiff differential equations; low order method

Syntax

```
[T,Y] = solver(odefun,tspan,y0)
[T,Y] = solver(odefun,tspan,y0,options)
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
sol = solver(odefun,[t0 tf],y0...)
```

This page contains an overview of the solver functions: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. You can call any of these solvers by substituting the placeholder, *`solver`*, with any of the function names.

Arguments

The following table describes the input arguments to the solvers.

`odefun`	A function handle that evaluates the right side of the differential equations. All solvers solve systems of equations in the form $y' = f(t,y)$ or problems that involve a mass matrix, $M(t,y)y' = f(t,y)$. The `ode23s` solver can solve only equations with constant mass matrices. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`. For `tspan` vectors with two elements `[t0 tf]`, the solver returns the solution evaluated at every integration step. For `tspan` vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.

	Specifying `tspan` with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from `tspan(1)` to `tspan(end)`. All solvers in the ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in `tspan`, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points. Specifying `tspan` with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management.
`y0`	A vector of initial conditions.
`options`	Structure of optional parameters that change the default integration properties. This is the fourth input argument. `[t,y] = solver(odefun,tspan,y0,options)` You can create options using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for the solvers.

`T`	Column vector of time points.
`Y`	Solution array. Each row in `Y` corresponds to the solution at a time returned in the corresponding row of `T`.
`TE`	The time at which an event occurs.
`YE`	The solution at the time of the event.

`IE`	The index `i` of the event function that vanishes.
`sol`	Structure to evaluate the solution.

Description

`[T,Y] = solver(odefun,tspan,y0)` with `tspan = [t0 tf]` integrates the system of differential equations $y' = f(t,y)$ from time `t0` to `tf` with initial conditions `y0`. The first input argument, `odefun`, is a function handle. The function, `f = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `f` corresponding to $f(t,y)$. Each row in the solution array `Y` corresponds to a time returned in column vector `T`. To obtain solutions at the specific times `t0`, `t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`[T,Y] = solver(odefun,tspan,y0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used properties include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components are `1e-6` by default). If certain components of the solution must be nonnegative, use the `odeset` function to set the `NonNegative` property to the indices of these components. See `odeset` for details.

`[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)` solves as above while also finding where functions of (t,y), called event functions, are zero. For each event function, you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, e.g., `events` or `@events`, and creating a function `[value,isterminal,direction] = events(t,y)`. For the `i`th event function in `events`,

- `value(i)` is the value of the function.
- `isterminal(i) = 1`, if the integration is to terminate at a zero of this event function and `0` otherwise.

- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Corresponding entries in `TE`, `YE`, and `IE` return, respectively, the time at which an event occurs, the solution at the time of the event, and the index `i` of the event function that vanishes.

`sol = `*`solver`*`(odefun,[t0 tf],y0...)` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0,tf]`. You must pass `odefun` as a function handle. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver.
`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.
`sol.solver`	Solver name.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen.

By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

For the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, the Jacobian matrix $\partial f/\partial y$ is critical to reliability and efficiency. Use `odeset` to set `Jacobian` to `@FJAC` if `FJAC(T,Y)` returns the Jacobian $\partial f/\partial y$ or to the matrix $\partial f/\partial y$ if the Jacobian is constant. If the `Jacobian` property is not set (the default), $\partial f/\partial y$ is approximated by finite differences. Set the `Vectorized` property 'on' if the ODE function is coded so that `odefun(T,[Y1,Y2 ...])` returns `[odefun(T,Y1),odefun(T,Y2) ...]`. If $\partial f/\partial y$ is a sparse matrix, set the `JPattern` property to the sparsity pattern of $\partial f/\partial y$, i.e., a sparse matrix `S` with `S(i,j) = 1` if the `i`th component of $f(t,y)$ depends on the `j`th component of y, and 0 otherwise.

The solvers of the ODE suite can solve problems of the form $M(t,y)y' = f(t,y)$, with time- and state-dependent mass matrix M. (The `ode23s` solver can solve only equations with constant mass matrices.) If a problem has a mass matrix, create a function `M = MASS(t,y)` that returns the value of the mass matrix, and use `odeset` to set the `Mass` property to `@MASS`. If the mass matrix is constant, the matrix should be used as the value of the `Mass` property. Problems with state-dependent mass matrices are more difficult:

- If the mass matrix does not depend on the state variable y and the function `MASS` is to be called with one input argument, `t`, set the `MStateDependence` property to '`none`'.
- If the mass matrix depends weakly on y, set `MStateDependence` to '`weak`' (the default); otherwise, set it to '`strong`'. In either case, the function `MASS` is called with the two arguments `(t,y)`.

If there are many differential equations, it is important to exploit sparsity:

- Return a sparse $M(t,y)$.

- Supply the sparsity pattern of $\partial f/\partial y$ using the `JPattern` property or a sparse $\partial f/\partial y$ using the `Jacobian` property.
- For strongly state-dependent $M(t,y)$, set `MvPattern` to a sparse matrix `S` with `S(i,j) = 1` if for any `k`, the `(i,k)` component of $M(t,y)$ depends on component `j` of y, and `0` otherwise.

If the mass matrix M is singular, then $M(t,y)y' = f(t,y)$ is a system of differential algebraic equations. DAEs have solutions only when y_0 is consistent, that is, if there is a vector yp_0 such that $M(t_0,y_0)yp_0 = f(t_0,y_0)$. The `ode15s` and `ode23t` solvers can solve DAEs of index 1 provided that `y0` is sufficiently close to being consistent. If there is a mass matrix, you can use `odeset` to set the `MassSingular` property to `'yes'`, `'no'`, or `'maybe'`. The default value of `'maybe'` causes the solver to test whether the problem is a DAE. You can provide `yp0` as the value of the `InitialSlope` property. The default is the zero vector. If a problem is a DAE, and `y0` and `yp0` are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. When solving DAEs, it is very advantageous to formulate the problem so that M is a diagonal matrix (a semi-explicit DAE).

Solver	Problem Type	Order of Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.
`ode23`	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.

Solver	Problem Type	Order of Accuracy	When to Use
`ode113`	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.
`ode15s`	Stiff	Low to medium	If `ode45` is slow because the problem is stiff.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
`ode23t`	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
`ode23tb`	Stiff	Low	If using crude error tolerances to solve stiff systems.

The algorithms used in the ODE solvers vary according to order of accuracy [6] and the type of systems (stiff or nonstiff) they are designed to solve. See "Algorithms" on page 1-3715 for more details.

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and "Integrator Options" in the MATLAB Mathematics documentation.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
RelTol, AbsTol, NormControl	√	√	√	√	√	√	√
OutputFcn, OutputSel, Refine, Stats	√	√	√	√	√	√	√
NonNegative	√	√	√	√ *	—	√ *	√ *
Events	√	√	√	√	√	√	√
MaxStep, InitialStep	√	√	√	√	√	√	√
Jacobian, JPattern, Vectorized	—	—	—	√	√	√	√
Mass MStateDependence MvPattern MassSingular	√ √ — —	√ √ — —	√ √ — —	√ √ √ √	√ — — —	√ √ √ √	√ √ √ —
InitialSlope	—	—	—	√	—	√	—
MaxOrder, BDF	—	—	—	√	—	—	—

Note You can use the `NonNegative` parameter with `ode15s`, `ode23t`, and `ode23tb` only for those problems for which there is no mass matrix.

Examples

Example 1

An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces.

$$
\begin{aligned}
y_1' &= y_2 y_3 & y_1(0) &= 0 \\
y_2' &= -y_1 y_3 & y_2(0) &= 1 \\
y_3' &= -0.51 y_1 y_2 & y_3(0) &= 1
\end{aligned}
$$

To simulate this system, create a function `rigid` containing the equations

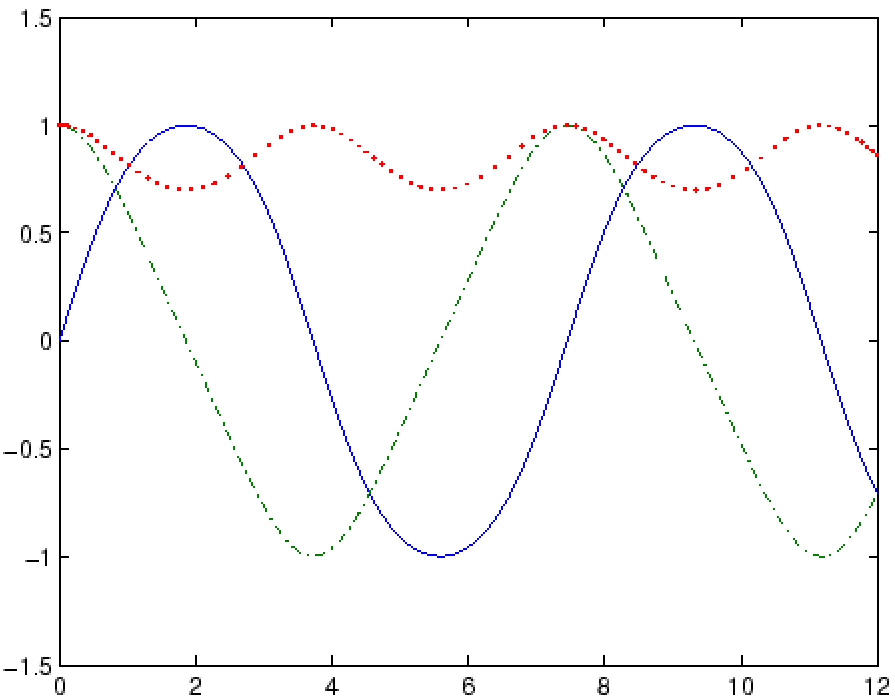
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time `0`.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[T,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

Plotting the columns of the returned array `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'.')
```



Example 2

An example of a stiff system is provided by the van der Pol equations in relaxation oscillation. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$
\begin{aligned}
y_1' &= y_2 & \quad y_1(0) &= 2 \\
y_2' &= 1000(1 - y_1^2)y_2 - y_1 & \quad y_2(0) &= 0
\end{aligned}
$$

To simulate this system, create a function `vdp1000` containing the equations

```
function dy = vdp1000(t,y)
```

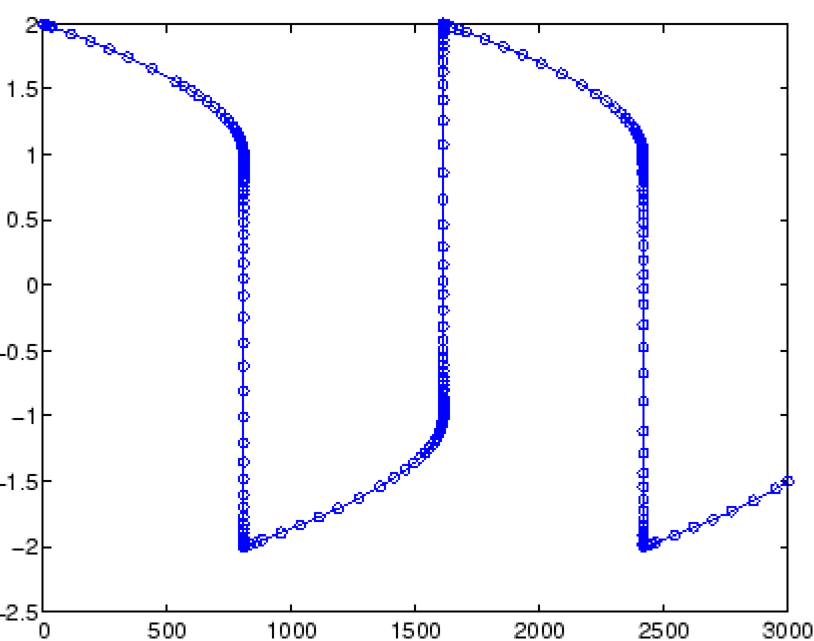
```
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

For this problem, we will use the default relative and absolute tolerances (`1e-3` and `1e-6`, respectively) and solve on a time interval of `[0 3000]` with initial condition vector `[2 0]` at time `0`.

```
[T,Y] = ode15s(@vdp1000,[0 3000],[2 0]);
```

Plotting the first column of the returned matrix `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-o')
```



Example 3

This example solves an ordinary differential equation with time-dependent terms.

Consider the following ODE, with time-dependent parameters defined only through the set of data points given in two vectors:

```
y'(t) + f(t)y(t) = g(t)
```

The initial condition is `y(0) = 0`, where the function `f(t)` is defined through the `n`-by-`1` vectors `tf` and `f`, and the function `g(t)` is defined through the `m`-by-`1` vectors `tg` and `g`.

First, define the time-dependent parameters `f(t)` and `g(t)` as the following:

```
ft = linspace(0,5,25); % Generate t for f
f = ft.^2 - ft - 3; % Generate f(t)
gt = linspace(1,6,25); % Generate t for g
g = 3*sin(gt-0.25); % Generate g(t)
```

Write a function to interpolate the data sets specified above to obtain the value of the time-dependent terms at the specified time:

```
function dydt = myode(t,y,ft,f,gt,g)
f = interp1(ft,f,t); % Interpolate the data set (ft,f) at time t
g = interp1(gt,g,t); % Interpolate the data set (gt,g) at time t
dydt = -f.*y + g; % Evalute ODE at time t
```

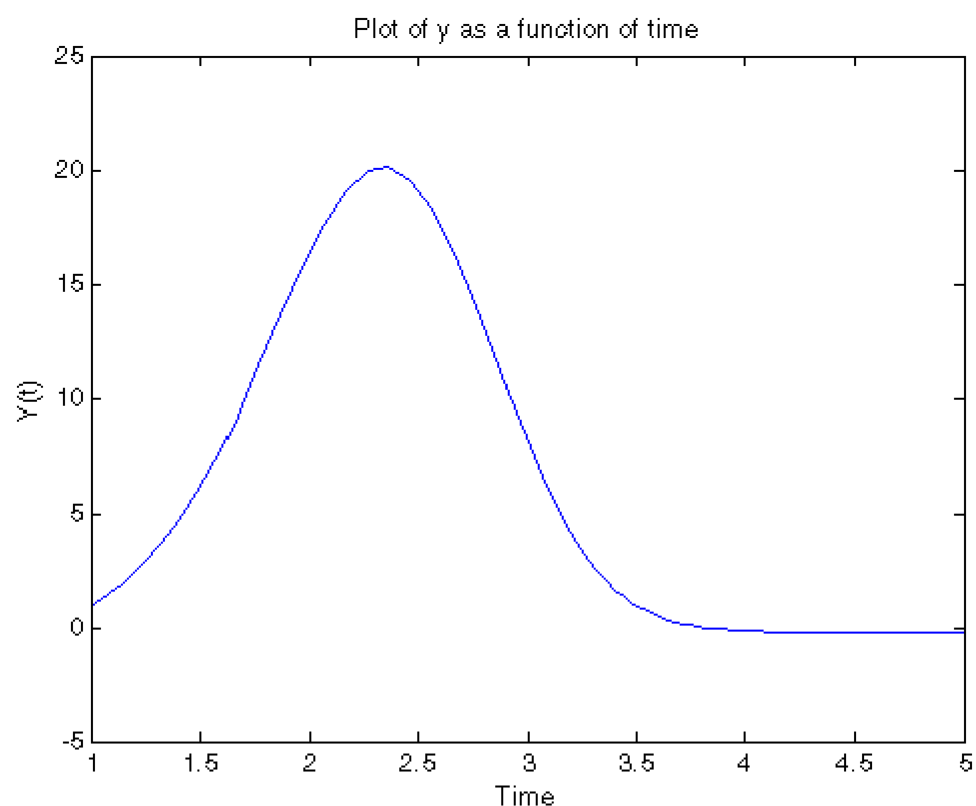
Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
Tspan = [1 5]; % Solve from t=1 to t=5
IC = 1; % y(t=0) = 1
[T Y] = ode45(@(t,y) myode(t,y,ft,f,gt,g),Tspan,IC); % Solve ODE
```

Plot the solution `y(t)` as a function of time:

```
plot(T, Y);
```

```
title('Plot of y as a function of time');
xlabel('Time'); ylabel('Y(t)');
```



Algorithms

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing `y(t`$_n$`)`, it needs only the solution at the immediately preceding time point, `y(t`$_{n-1}$`)`. In general, `ode45` is the best function to apply as a *first try* for most problems. [3]

`ode23` is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude

tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver. [2]

`ode113` is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE file function is particularly expensive to evaluate. `ode113` is a *multistep* solver — it normally needs the solutions at several preceding time points to compute the current solution. [7]

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` when `ode45` fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem. [9], [10]

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective. [9]

`ode23t` is an implementation of the trapezoidal rule using a "free" interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve DAEs. [10]

`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances. [8], [1]

References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R. Smith, "Transient Simulation of Silicon Devices and Circuits," *IEEE Trans. CAD*, 4 (1985), pp. 436–451.

[2] Bogacki, P. and L. F. Shampine, “A 3(2) pair of Runge-Kutta formulas,” *Appl. Math. Letters*, Vol. 2, 1989, pp. 321–325.

[3] Dormand, J. R. and P. J. Prince, “A family of embedded Runge-Kutta formulae,” *J. Comp. Appl. Math.*, Vol. 6, 1980, pp. 19–26.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, San Francisco, 1975.

[8] Shampine, L. F. and M. E. Hosea, “Analysis and Implementation of TR-BDF2,” *Applied Numerical Mathematics 20*, 1996.

[9] Shampine, L. F. and M. W. Reichelt, “The MATLAB ODE Suite,” *SIAM Journal on Scientific Computing,* Vol. 18, 1997, pp. 1–22.

[10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, “Solving Index-1 DAEs in MATLAB and Simulink,” *SIAM Review*, Vol. 41, 1999, pp. 538–552.

See Also

`deval` | `ode15i` | `odeget` | `odeset` | `function_handle`

ode45

Purpose

Solve nonstiff differential equations; medium order method

Syntax

```
[T,Y] = solver(odefun,tspan,y0)
[T,Y] = solver(odefun,tspan,y0,options)
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
sol = solver(odefun,[t0 tf],y0...)
```

This page contains an overview of the solver functions: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. You can call any of these solvers by substituting the placeholder, *`solver`*, with any of the function names.

Arguments

The following table describes the input arguments to the solvers.

`odefun`	A function handle that evaluates the right side of the differential equations. All solvers solve systems of equations in the form $y' = f(t,y)$ or problems that involve a mass matrix, $M(t,y)y' = f(t,y)$. The `ode23s` solver can solve only equations with constant mass matrices. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`. For `tspan` vectors with two elements `[t0 tf]`, the solver returns the solution evaluated at every integration step. For `tspan` vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.

	Specifying `tspan` with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from `tspan(1)` to `tspan(end)`. All solvers in the ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in `tspan`, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points. Specifying `tspan` with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management.
`y0`	A vector of initial conditions.
`options`	Structure of optional parameters that change the default integration properties. This is the fourth input argument. `[t,y] = solver(odefun,tspan,y0,options)` You can create options using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for the solvers.

`T`	Column vector of time points.
`Y`	Solution array. Each row in `Y` corresponds to the solution at a time returned in the corresponding row of `T`.
`TE`	The time at which an event occurs.
`YE`	The solution at the time of the event.

`IE`	The index `i` of the event function that vanishes.
`sol`	Structure to evaluate the solution.

Description

`[T,Y] = `*`solver`*`(odefun,tspan,y0)` with `tspan = [t0 tf]` integrates the system of differential equations $y' = f(t,y)$ from time `t0` to `tf` with initial conditions `y0`. The first input argument, `odefun`, is a function handle. The function, `f = odefun(t,y)`, for a scalar `t` and a column vector `y`, must return a column vector `f` corresponding to $f(t,y)$. Each row in the solution array `Y` corresponds to a time returned in column vector `T`. To obtain solutions at the specific times `t0`, `t1,...,tf` (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`[T,Y] = `*`solver`*`(odefun,tspan,y0,options)` solves as above with default integration parameters replaced by property values specified in `options`, an argument created with the `odeset` function. Commonly used properties include a scalar relative error tolerance `RelTol` (`1e-3` by default) and a vector of absolute error tolerances `AbsTol` (all components are `1e-6` by default). If certain components of the solution must be nonnegative, use the `odeset` function to set the `NonNegative` property to the indices of these components. See `odeset` for details.

`[T,Y,TE,YE,IE] = `*`solver`*`(odefun,tspan,y0,options)` solves as above while also finding where functions of (t,y), called event functions, are zero. For each event function, you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the `'Events'` property to a function, e.g., `events` or `@events`, and creating a function `[value,isterminal,direction] = events(t,y)`. For the `i`th event function in `events`,

- `value(i)` is the value of the function.
- `isterminal(i) = 1`, if the integration is to terminate at a zero of this event function and `0` otherwise.

- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Corresponding entries in `TE`, `YE`, and `IE` return, respectively, the time at which an event occurs, the solution at the time of the event, and the index `i` of the event function that vanishes.

`sol = `*`solver`*`(odefun,[t0 tf],y0...)` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0,tf]`. You must pass `odefun` as a function handle. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver.
`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.
`sol.solver`	Solver name.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen.

By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

For the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, the Jacobian matrix $\partial f/\partial y$ is critical to reliability and efficiency. Use `odeset` to set `Jacobian` to `@FJAC` if `FJAC(T,Y)` returns the Jacobian $\partial f/\partial y$ or to the matrix $\partial f/\partial y$ if the Jacobian is constant. If the `Jacobian` property is not set (the default), $\partial f/\partial y$ is approximated by finite differences. Set the `Vectorized` property `'on'` if the ODE function is coded so that `odefun(T,[Y1,Y2 ...])` returns `[odefun(T,Y1),odefun(T,Y2) ...]`. If $\partial f/\partial y$ is a sparse matrix, set the `JPattern` property to the sparsity pattern of $\partial f/\partial y$, i.e., a sparse matrix `S` with `S(i,j) = 1` if the `i`th component of $f(t,y)$ depends on the `j`th component of y, and 0 otherwise.

The solvers of the ODE suite can solve problems of the form $M(t,y)y' = f(t,y)$, with time- and state-dependent mass matrix M. (The `ode23s` solver can solve only equations with constant mass matrices.) If a problem has a mass matrix, create a function `M = MASS(t,y)` that returns the value of the mass matrix, and use `odeset` to set the `Mass` property to `@MASS`. If the mass matrix is constant, the matrix should be used as the value of the `Mass` property. Problems with state-dependent mass matrices are more difficult:

- If the mass matrix does not depend on the state variable y and the function `MASS` is to be called with one input argument, `t`, set the `MStateDependence` property to `'none'`.
- If the mass matrix depends weakly on y, set `MStateDependence` to `'weak'` (the default); otherwise, set it to `'strong'`. In either case, the function `MASS` is called with the two arguments `(t,y)`.

If there are many differential equations, it is important to exploit sparsity:

- Return a sparse $M(t,y)$.

- Supply the sparsity pattern of $\partial f/\partial y$ using the `JPattern` property or a sparse $\partial f/\partial y$ using the `Jacobian` property.
- For strongly state-dependent $M(t,y)$, set `MvPattern` to a sparse matrix `S` with `S(i,j) = 1` if for any `k`, the `(i,k)` component of $M(t,y)$ depends on component `j` of y, and `0` otherwise.

If the mass matrix M is singular, then $M(t,y)y' = f(t,y)$ is a system of differential algebraic equations. DAEs have solutions only when y_0 is consistent, that is, if there is a vector yp_0 such that $M(t_0,y_0)yp_0 = f(t_0,y_0)$. The `ode15s` and `ode23t` solvers can solve DAEs of index 1 provided that `y0` is sufficiently close to being consistent. If there is a mass matrix, you can use `odeset` to set the `MassSingular` property to `'yes'`, `'no'`, or `'maybe'`. The default value of `'maybe'` causes the solver to test whether the problem is a DAE. You can provide `yp0` as the value of the `InitialSlope` property. The default is the zero vector. If a problem is a DAE, and `y0` and `yp0` are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. When solving DAEs, it is very advantageous to formulate the problem so that M is a diagonal matrix (a semi-explicit DAE).

Solver	Problem Type	Order of Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.
`ode23`	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.

Solver	Problem Type	Order of Accuracy	When to Use
`ode113`	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.
`ode15s`	Stiff	Low to medium	If `ode45` is slow because the problem is stiff.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
`ode23t`	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
`ode23tb`	Stiff	Low	If using crude error tolerances to solve stiff systems.

The algorithms used in the ODE solvers vary according to order of accuracy [6] and the type of systems (stiff or nonstiff) they are designed to solve. See “Algorithms” on page 1-3730 for more details.

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and “Integrator Options” in the MATLAB Mathematics documentation.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
RelTol, AbsTol, NormControl	√	√	√	√	√	√	√
OutputFcn, OutputSel, Refine, Stats	√	√	√	√	√	√	√
NonNegative	√	√	√	√ *	—	√ *	√ *
Events	√	√	√	√	√	√	√
MaxStep, InitialStep	√	√	√	√	√	√	√
Jacobian, JPattern, Vectorized	—	—	—	√	√	√	√
Mass MStateDependence MvPattern MassSingular	√ √ — —	√ √ — —	√ √ — —	√ √ √ √	√ — — —	√ √ √ √	√ √ √ —
InitialSlope	—	—	—	√	—	√	—
MaxOrder, BDF	—	—	—	√	—	—	—

Note You can use the `NonNegative` parameter with `ode15s`, `ode23t`, and `ode23tb` only for those problems for which there is no mass matrix.

Examples

Example 1

An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces.

$$y_1' = y_2 y_3 \qquad y_1(0) = 0$$
$$y_2' = -y_1 y_3 \qquad y_2(0) = 1$$
$$y_3' = -0.51 y_1 y_2 \qquad y_3(0) = 1$$

To simulate this system, create a function `rigid` containing the equations

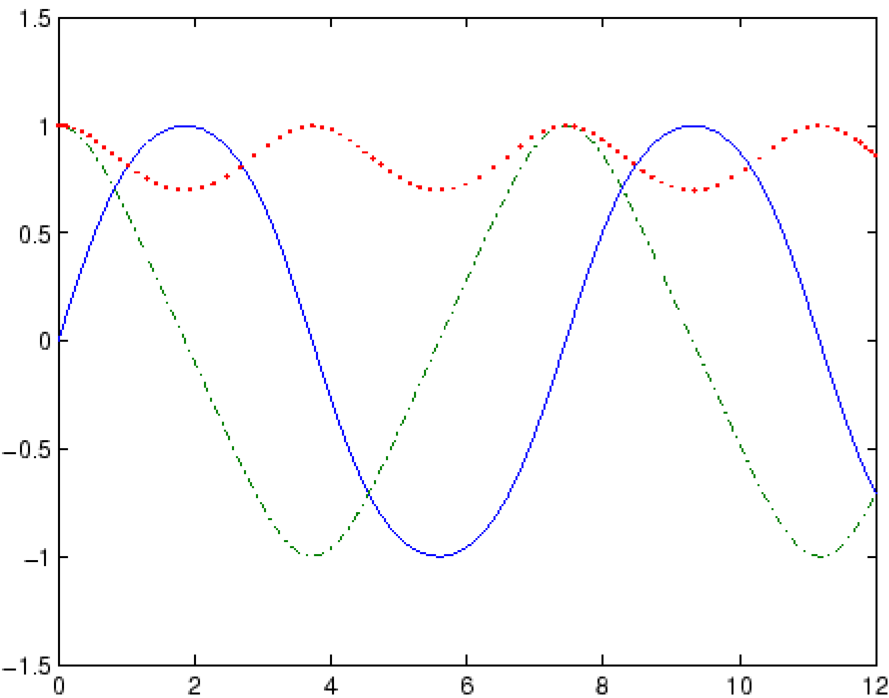
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time `0`.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[T,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

Plotting the columns of the returned array `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'.')
```



Example 2

An example of a stiff system is provided by the van der Pol equations in relaxation oscillation. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$
\begin{aligned}
y_1' &= y_2 & y_1(0) &= 2 \\
y_2' &= 1000(1 - y_1^2)y_2 - y_1 & y_2(0) &= 0
\end{aligned}
$$

To simulate this system, create a function `vdp1000` containing the equations

```
function dy = vdp1000(t,y)
```

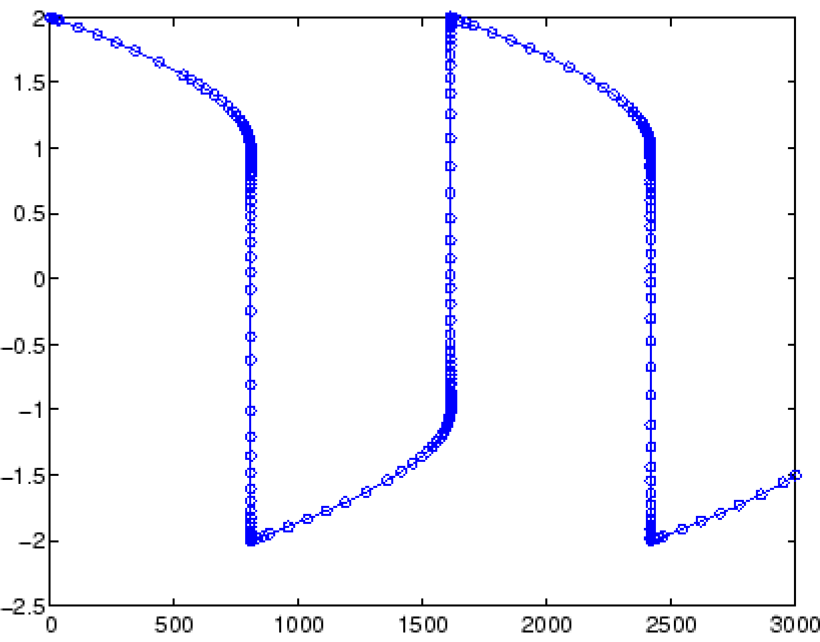
```
dy = zeros(2,1);    % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

For this problem, we will use the default relative and absolute tolerances (`1e-3` and `1e-6`, respectively) and solve on a time interval of `[0 3000]` with initial condition vector `[2 0]` at time `0`.

```
[T,Y] = ode15s(@vdp1000,[0 3000],[2 0]);
```

Plotting the first column of the returned matrix `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-o')
```



Example 3

This example solves an ordinary differential equation with time-dependent terms.

Consider the following ODE, with time-dependent parameters defined only through the set of data points given in two vectors:

```
y'(t) + f(t)y(t) = g(t)
```

The initial condition is `y(0) = 0`, where the function `f(t)` is defined through the `n`-by-`1` vectors `tf` and `f`, and the function `g(t)` is defined through the `m`-by-`1` vectors `tg` and `g`.

First, define the time-dependent parameters `f(t)` and `g(t)` as the following:

```
ft = linspace(0,5,25); % Generate t for f
f = ft.^2 - ft - 3; % Generate f(t)
gt = linspace(1,6,25); % Generate t for g
g = 3*sin(gt-0.25); % Generate g(t)
```

Write a function to interpolate the data sets specified above to obtain the value of the time-dependent terms at the specified time:

```
function dydt = myode(t,y,ft,f,gt,g)
f = interp1(ft,f,t); % Interpolate the data set (ft,f) at time t
g = interp1(gt,g,t); % Interpolate the data set (gt,g) at time t
dydt = -f.*y + g; % Evalute ODE at time t
```

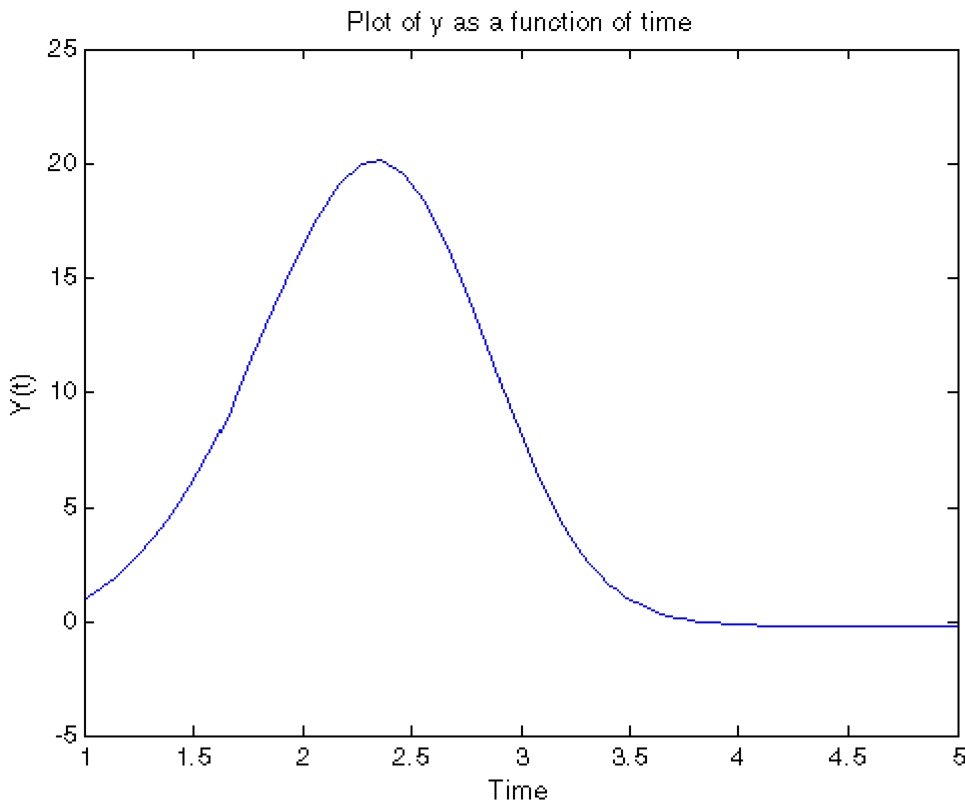
Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
Tspan = [1 5]; % Solve from t=1 to t=5
IC = 1; % y(t=0) = 1
[T Y] = ode45(@(t,y) myode(t,y,ft,f,gt,g),Tspan,IC); % Solve ODE
```

Plot the solution `y(t)` as a function of time:

```
plot(T, Y);
```

```
title('Plot of y as a function of time');
xlabel('Time'); ylabel('Y(t)');
```



Algorithms

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing `y(`t_n`)`, it needs only the solution at the immediately preceding time point, `y(`t_{n-1}`)`. In general, `ode45` is the best function to apply as a *first try* for most problems. [3]

`ode23` is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude

tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver. [2]

`ode113` is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE file function is particularly expensive to evaluate. `ode113` is a *multistep* solver — it normally needs the solutions at several preceding time points to compute the current solution. [7]

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` when `ode45` fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem. [9], [10]

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective. [9]

`ode23t` is an implementation of the trapezoidal rule using a "free" interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve DAEs. [10]

`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances. [8], [1]

References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R. Smith, "Transient Simulation of Silicon Devices and Circuits," *IEEE Trans. CAD*, 4 (1985), pp. 436–451.

[2] Bogacki, P. and L. F. Shampine, “A 3(2) pair of Runge-Kutta formulas,” *Appl. Math. Letters*, Vol. 2, 1989, pp. 321–325.

[3] Dormand, J. R. and P. J. Prince, “A family of embedded Runge-Kutta formulae,” *J. Comp. Appl. Math.*, Vol. 6, 1980, pp. 19–26.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, San Francisco, 1975.

[8] Shampine, L. F. and M. E. Hosea, “Analysis and Implementation of TR-BDF2,” *Applied Numerical Mathematics 20*, 1996.

[9] Shampine, L. F. and M. W. Reichelt, “The MATLAB ODE Suite,” *SIAM Journal on Scientific Computing,* Vol. 18, 1997, pp. 1–22.

[10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, “Solving Index-1 DAEs in MATLAB and Simulink,” *SIAM Review*, Vol. 41, 1999, pp. 538–552.

See Also

`deval` | `ode15i` | `odeget` | `odeset` | `function_handle`

Purpose

Solve nonstiff differential equations; variable order method

Syntax

```
[T,Y] = solver(odefun,tspan,y0)
[T,Y] = solver(odefun,tspan,y0,options)
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
sol = solver(odefun,[t0 tf],y0...)
```

This page contains an overview of the solver functions: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. You can call any of these solvers by substituting the placeholder, *`solver`*, with any of the function names.

Arguments

The following table describes the input arguments to the solvers.

`odefun`	A function handle that evaluates the right side of the differential equations. All solvers solve systems of equations in the form $y' = f(t,y)$ or problems that involve a mass matrix, $M(t,y)y' = f(t,y)$. The `ode23s` solver can solve only equations with constant mass matrices. `ode15s` and `ode23t` can solve problems with a mass matrix that is singular, i.e., differential-algebraic equations (DAEs).
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`. For `tspan` vectors with two elements `[t0 tf]`, the solver returns the solution evaluated at every integration step. For `tspan` vectors with more than two elements, the solver returns solutions evaluated at the given time points. The time values must be in order, either increasing or decreasing.

	Specifying `tspan` with more than two elements does not affect the internal time steps that the solver uses to traverse the interval from `tspan(1)` to `tspan(end)`. All solvers in the ODE suite obtain output values by means of continuous extensions of the basic formulas. Although a solver does not necessarily step precisely to a time point specified in `tspan`, the solutions produced at the specified time points are of the same order of accuracy as the solutions computed at the internal time points. Specifying `tspan` with more than two elements has little effect on the efficiency of computation, but for large systems, affects memory management.
`y0`	A vector of initial conditions.
`options`	Structure of optional parameters that change the default integration properties. This is the fourth input argument. `[t,y] = solver(odefun,tspan,y0,options)` You can create options using the `odeset` function. See `odeset` for details.

The following table lists the output arguments for the solvers.

`T`	Column vector of time points.
`Y`	Solution array. Each row in `Y` corresponds to the solution at a time returned in the corresponding row of `T`.
`TE`	The time at which an event occurs.
`YE`	The solution at the time of the event.

IE	The index i of the event function that vanishes.
sol	Structure to evaluate the solution.

Description

[T,Y] = *solver*(odefun,tspan,y0) with tspan = [t0 tf] integrates the system of differential equations $y' = f(t,y)$ from time t0 to tf with initial conditions y0. The first input argument, odefun, is a function handle. The function, f = odefun(t,y), for a scalar t and a column vector y, must return a column vector f corresponding to $f(t,y)$. Each row in the solution array Y corresponds to a time returned in column vector T. To obtain solutions at the specific times t0, t1,...,tf (all increasing or all decreasing), use tspan = [t0,t1,...,tf].

"Parameterizing Functions" explains how to provide additional parameters to the function fun, if necessary.

[T,Y] = *solver*(odefun,tspan,y0,options) solves as above with default integration parameters replaced by property values specified in options, an argument created with the odeset function. Commonly used properties include a scalar relative error tolerance RelTol (1e-3 by default) and a vector of absolute error tolerances AbsTol (all components are 1e-6 by default). If certain components of the solution must be nonnegative, use the odeset function to set the NonNegative property to the indices of these components. See odeset for details.

[T,Y,TE,YE,IE] = *solver*(odefun,tspan,y0,options) solves as above while also finding where functions of (t,y), called event functions, are zero. For each event function, you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Do this by setting the 'Events' property to a function, e.g., events or @events, and creating a function [value,isterminal,direction] = events(t,y). For the ith event function in events,

- value(i) is the value of the function.
- isterminal(i) = 1, if the integration is to terminate at a zero of this event function and 0 otherwise.

- `direction(i) = 0` if all zeros are to be computed (the default), `+1` if only the zeros where the event function increases, and `-1` if only the zeros where the event function decreases.

Corresponding entries in `TE`, `YE`, and `IE` return, respectively, the time at which an event occurs, the solution at the time of the event, and the index `i` of the event function that vanishes.

`sol = solver(odefun,[t0 tf],y0...)` returns a structure that you can use with `deval` to evaluate the solution at any point on the interval `[t0,tf]`. You must pass `odefun` as a function handle. The structure `sol` always includes these fields:

`sol.x`	Steps chosen by the solver.
`sol.y`	Each column `sol.y(:,i)` contains the solution at `sol.x(i)`.
`sol.solver`	Solver name.

If you specify the `Events` option and events are detected, `sol` also includes these fields:

`sol.xe`	Points at which events, if any, occurred. `sol.xe(end)` contains the exact point of a terminal event, if any.
`sol.ye`	Solutions that correspond to events in `sol.xe`.
`sol.ie`	Indices into the vector returned by the function specified in the `Events` option. The values indicate which event the solver detected.

If you specify an output function as the value of the `OutputFcn` property, the solver calls it with the computed solution after each time step. Four output functions are provided: `odeplot`, `odephas2`, `odephas3`, `odeprint`. When you call the solver with no output arguments, it calls the default `odeplot` to plot the solution as it is computed. `odephas2` and `odephas3` produce two- and three-dimensional phase plane plots, respectively. `odeprint` displays the solution components on the screen.

By default, the ODE solver passes all components of the solution to the output function. You can pass only specific components by providing a vector of indices as the value of the `OutputSel` property. For example, if you call the solver with no output arguments and set the value of `OutputSel` to `[1,3]`, the solver plots solution components 1 and 3 as they are computed.

For the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`, the Jacobian matrix $\partial f/\partial y$ is critical to reliability and efficiency. Use `odeset` to set `Jacobian` to `@FJAC` if `FJAC(T,Y)` returns the Jacobian $\partial f/\partial y$ or to the matrix $\partial f/\partial y$ if the Jacobian is constant. If the `Jacobian` property is not set (the default), $\partial f/\partial y$ is approximated by finite differences. Set the `Vectorized` property 'on' if the ODE function is coded so that `odefun(T,[Y1,Y2 ...])` returns `[odefun(T,Y1),odefun(T,Y2) ...]`. If $\partial f/\partial y$ is a sparse matrix, set the `JPattern` property to the sparsity pattern of $\partial f/\partial y$, i.e., a sparse matrix `S` with `S(i,j) = 1` if the `i`th component of $f(t,y)$ depends on the `j`th component of y, and 0 otherwise.

The solvers of the ODE suite can solve problems of the form $M(t,y)y' = f(t,y)$, with time- and state-dependent mass matrix M. (The `ode23s` solver can solve only equations with constant mass matrices.) If a problem has a mass matrix, create a function `M = MASS(t,y)` that returns the value of the mass matrix, and use `odeset` to set the `Mass` property to `@MASS`. If the mass matrix is constant, the matrix should be used as the value of the `Mass` property. Problems with state-dependent mass matrices are more difficult:

- If the mass matrix does not depend on the state variable y and the function `MASS` is to be called with one input argument, `t`, set the `MStateDependence` property to '`none`'.
- If the mass matrix depends weakly on y, set `MStateDependence` to '`weak`' (the default); otherwise, set it to '`strong`'. In either case, the function `MASS` is called with the two arguments `(t,y)`.

If there are many differential equations, it is important to exploit sparsity:

- Return a sparse $M(t,y)$.

- Supply the sparsity pattern of $\partial f/\partial y$ using the `JPattern` property or a sparse $\partial f/\partial y$ using the `Jacobian` property.
- For strongly state-dependent $M(t,y)$, set `MvPattern` to a sparse matrix `S` with `S(i,j) = 1` if for any `k`, the `(i,k)` component of $M(t,y)$ depends on component `j` of y, and `0` otherwise.

If the mass matrix M is singular, then $M(t,y)y' = f(t,y)$ is a system of differential algebraic equations. DAEs have solutions only when y_0 is consistent, that is, if there is a vector yp_0 such that $M(t_0,y_0)yp_0 = f(t_0,y_0)$. The `ode15s` and `ode23t` solvers can solve DAEs of index 1 provided that `y0` is sufficiently close to being consistent. If there is a mass matrix, you can use `odeset` to set the `MassSingular` property to `'yes'`, `'no'`, or `'maybe'`. The default value of `'maybe'` causes the solver to test whether the problem is a DAE. You can provide `yp0` as the value of the `InitialSlope` property. The default is the zero vector. If a problem is a DAE, and `y0` and `yp0` are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve the problem. When solving DAEs, it is very advantageous to formulate the problem so that M is a diagonal matrix (a semi-explicit DAE).

Solver	Problem Type	Order of Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.
`ode23`	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.

Solver	Problem Type	Order of Accuracy	When to Use
`ode113`	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.
`ode15s`	Stiff	Low to medium	If `ode45` is slow because the problem is stiff.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
`ode23t`	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
`ode23tb`	Stiff	Low	If using crude error tolerances to solve stiff systems.

The algorithms used in the ODE solvers vary according to order of accuracy [6] and the type of systems (stiff or nonstiff) they are designed to solve. See “Algorithms” on page 1-3745 for more details.

Options

Different solvers accept different parameters in the options list. For more information, see `odeset` and “Integrator Options” in the MATLAB Mathematics documentation.

Parameters	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb
RelTol, AbsTol, NormControl	√	√	√	√	√	√	√
OutputFcn, OutputSel, Refine, Stats	√	√	√	√	√	√	√
NonNegative	√	√	√	√ *	—	√ *	√ *
Events	√	√	√	√	√	√	√
MaxStep, InitialStep	√	√	√	√	√	√	√
Jacobian, JPattern, Vectorized	—	—	—	√	√	√	√
Mass MStateDependence MvPattern MassSingular	√ √ — —	√ √ — —	√ √ — —	√ √ √ √	√ — — —	√ √ √ √	√ √ √ —
InitialSlope	—	—	—	√	—	√	—
MaxOrder, BDF	—	—	—	√	—	—	—

Note You can use the `NonNegative` parameter with `ode15s`, `ode23t`, and `ode23tb` only for those problems for which there is no mass matrix.

Examples

Example 1

An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces.

$$y'_1 = y_2 y_3 \qquad y_1(0) = 0$$
$$y'_2 = -y_1 y_3 \qquad y_2(0) = 1$$
$$y'_3 = -0.51 y_1 y_2 \qquad y_3(0) = 1$$

To simulate this system, create a function `rigid` containing the equations

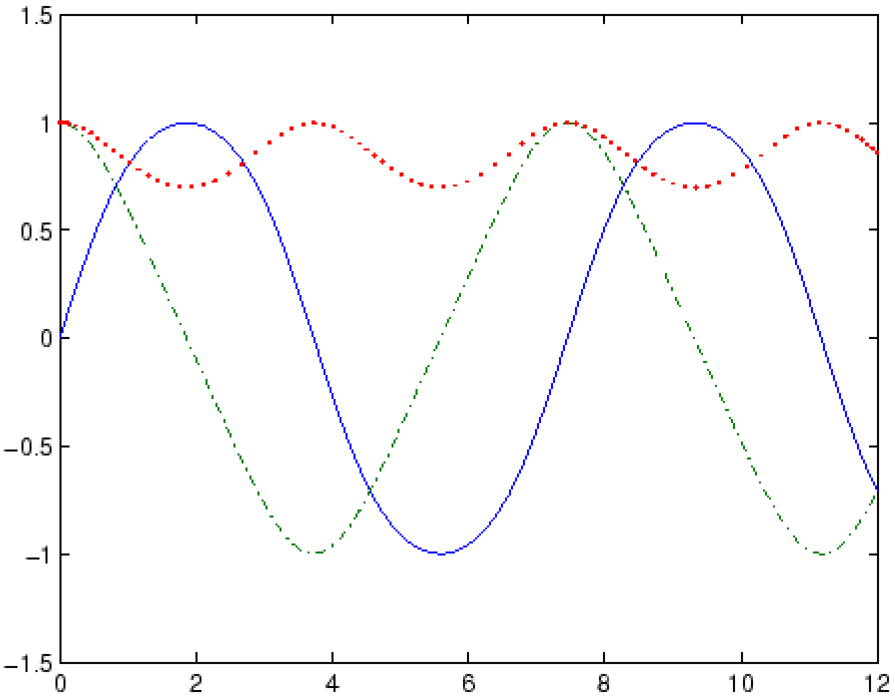
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time `0`.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[T,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

Plotting the columns of the returned array `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'.')
```



Example 2

An example of a stiff system is provided by the van der Pol equations in relaxation oscillation. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff.

$$
\begin{aligned}
y_1' &= y_2 & y_1(0) &= 2 \\
y_2' &= 1000(1 - y_1^2)y_2 - y_1 & y_2(0) &= 0
\end{aligned}
$$

To simulate this system, create a function `vdp1000` containing the equations

```
function dy = vdp1000(t,y)
```

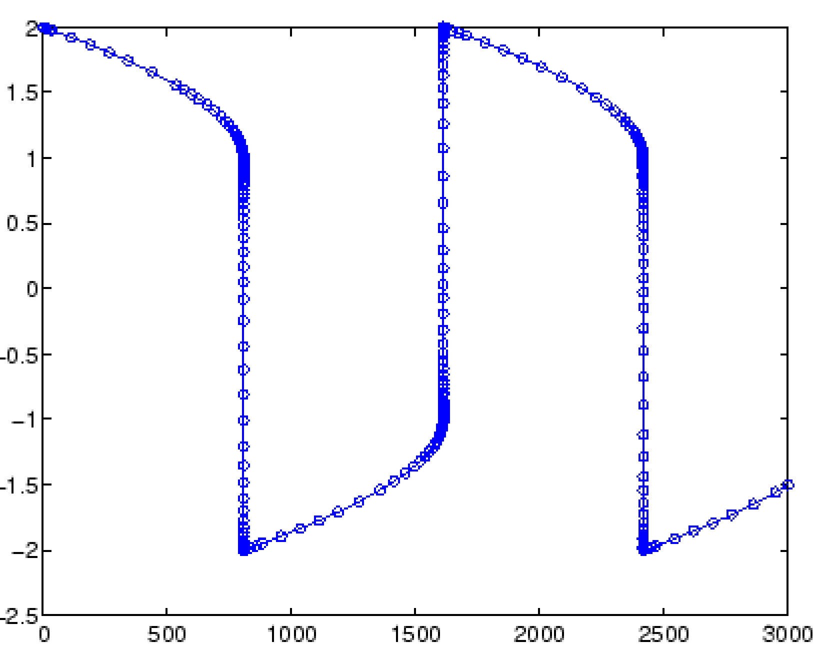
```
dy = zeros(2,1);     % a column vector
dy(1) = y(2);
dy(2) = 1000*(1 - y(1)^2)*y(2) - y(1);
```

For this problem, we will use the default relative and absolute tolerances (`1e-3` and `1e-6`, respectively) and solve on a time interval of `[0 3000]` with initial condition vector `[2 0]` at time `0`.

```
[T,Y] = ode15s(@vdp1000,[0 3000],[2 0]);
```

Plotting the first column of the returned matrix `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-o')
```



Example 3

This example solves an ordinary differential equation with time-dependent terms.

Consider the following ODE, with time-dependent parameters defined only through the set of data points given in two vectors:

```
y'(t) + f(t)y(t) = g(t)
```

The initial condition is `y(0) = 0`, where the function `f(t)` is defined through the `n`-by-`1` vectors `tf` and `f`, and the function `g(t)` is defined through the `m`-by-`1` vectors `tg` and `g`.

First, define the time-dependent parameters `f(t)` and `g(t)` as the following:

```
ft = linspace(0,5,25); % Generate t for f
f = ft.^2 - ft - 3; % Generate f(t)
gt = linspace(1,6,25); % Generate t for g
g = 3*sin(gt-0.25); % Generate g(t)
```

Write a function to interpolate the data sets specified above to obtain the value of the time-dependent terms at the specified time:

```
function dydt = myode(t,y,ft,f,gt,g)
f = interp1(ft,f,t); % Interpolate the data set (ft,f) at time t
g = interp1(gt,g,t); % Interpolate the data set (gt,g) at time t
dydt = -f.*y + g; % Evalute ODE at time t
```

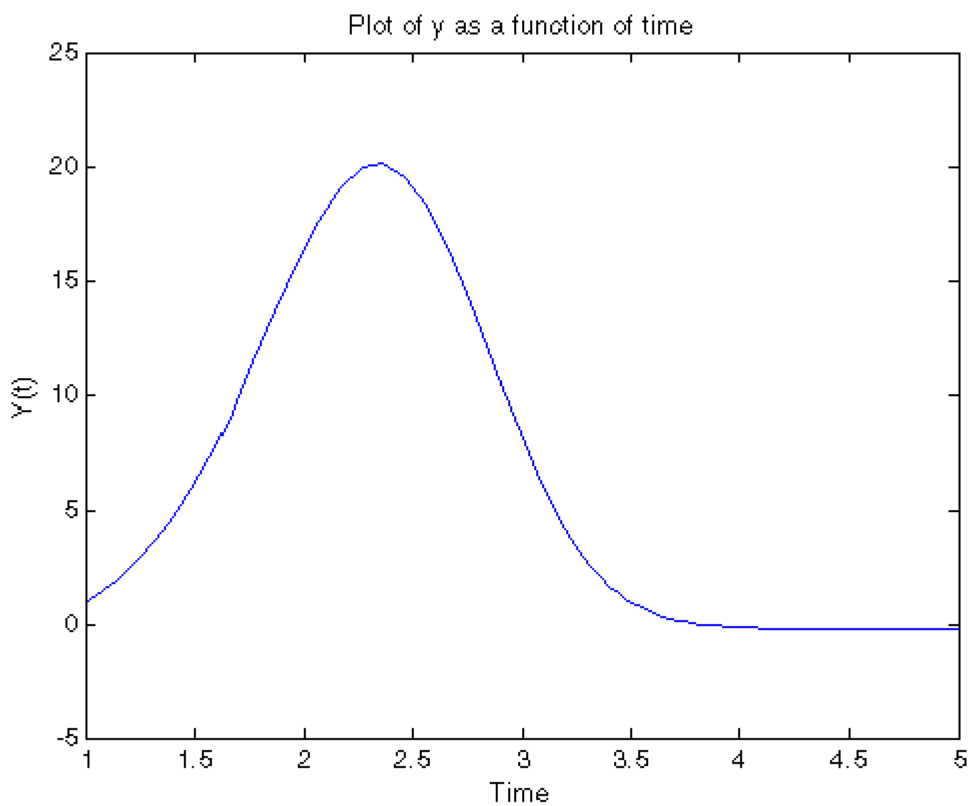
Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
Tspan = [1 5]; % Solve from t=1 to t=5
IC = 1; % y(t=0) = 1
[T Y] = ode45(@(t,y) myode(t,y,ft,f,gt,g),Tspan,IC); % Solve ODE
```

Plot the solution `y(t)` as a function of time:

```
plot(T, Y);
```

```
title('Plot of y as a function of time');
xlabel('Time'); ylabel('Y(t)');
```



Algorithms

`ode45` is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing $y(t_n)$, it needs only the solution at the immediately preceding time point, $y(t_{n-1})$. In general, `ode45` is the best function to apply as a *first try* for most problems. [3]

`ode23` is an implementation of an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude

tolerances and in the presence of moderate stiffness. Like `ode45`, `ode23` is a one-step solver. [2]

`ode113` is a variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE file function is particularly expensive to evaluate. `ode113` is a *multistep* solver — it normally needs the solutions at several preceding time points to compute the current solution. [7]

The above algorithms are intended to solve nonstiff systems. If they appear to be unduly slow, try using one of the stiff solvers below.

`ode15s` is a variable order solver based on the numerical differentiation formulas (NDFs). Optionally, it uses the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. Like `ode113`, `ode15s` is a multistep solver. Try `ode15s` when `ode45` fails, or is very inefficient, and you suspect that the problem is stiff, or when solving a differential-algebraic problem. [9], [10]

`ode23s` is based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective. [9]

`ode23t` is an implementation of the trapezoidal rule using a "free" interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve DAEs. [10]

`ode23tb` is an implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order two. By construction, the same iteration matrix is used in evaluating both stages. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances. [8], [1]

References

[1] Bank, R. E., W. C. Coughran, Jr., W. Fichtner, E. Grosse, D. Rose, and R. Smith, "Transient Simulation of Silicon Devices and Circuits," *IEEE Trans. CAD*, 4 (1985), pp. 436–451.

[2] Bogacki, P. and L. F. Shampine, "A 3(2) pair of Runge-Kutta formulas," *Appl. Math. Letters*, Vol. 2, 1989, pp. 321–325.

[3] Dormand, J. R. and P. J. Prince, "A family of embedded Runge-Kutta formulae," *J. Comp. Appl. Math.*, Vol. 6, 1980, pp. 19–26.

[4] Forsythe, G. , M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[5] Kahaner, D. , C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[6] Shampine, L. F. , *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[7] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, San Francisco, 1975.

[8] Shampine, L. F. and M. E. Hosea, "Analysis and Implementation of TR-BDF2," *Applied Numerical Mathematics 20*, 1996.

[9] Shampine, L. F. and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing,* Vol. 18, 1997, pp. 1–22.

[10] Shampine, L. F., M. W. Reichelt, and J.A. Kierzenka, "Solving Index-1 DAEs in MATLAB and Simulink," *SIAM Review*, Vol. 41, 1999, pp. 538–552.

See Also

`deval` | `ode15i` | `odeget` | `odeset` | `function_handle`

odeget

Purpose

Ordinary differential equation options parameters

Syntax

```
o = odeget(options,'name')
o = odeget(options,'name',default)
```

Description

`o = odeget(options,'name')` extracts the value of the property specified by string `'name'` from integrator options structure `options`, returning an empty matrix if the property value is not specified in `options`. It is only necessary to type the leading characters that uniquely identify the property name. Case is ignored for property names. The empty matrix `[]` is a valid `options` argument.

`o = odeget(options,'name',default)` returns `o = default` if the named property is not specified in `options`.

Examples

Having constructed an ODE options structure,

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-3 2e-3 3e-3]);
```

you can view these property settings with `odeget`.

```
odeget(options,'RelTol')
ans =

       1.0000e-04

odeget(options,'AbsTol')
ans =

    0.0010    0.0020    0.0030
```

See Also

`odeset`

Purpose

Create or alter options structure for ordinary differential equation solvers

Syntax

```
options = odeset('name1',value1,'name2',value2,...)
options = odeset(oldopts,'name1',value1,...)
options = odeset(oldopts,newopts)
odeset
```

Description

The `odeset` function lets you adjust the integration parameters of the following ODE solvers.

For solving fully implicit differential equations:

```
ode15i
```

For solving initial value problems:

```
ode23, ode45, ode113, ode15s, ode23s, ode23t, ode23tb
```

See below for information about the integration parameters.

`options = odeset('name1',value1,'name2',value2,...)` creates an options structure that you can pass as an argument to any of the ODE solvers. In the resulting structure, `options`, the named properties have the specified values. For example, `'name1'` has the value `value1`. Any unspecified properties have default values. It is sufficient to type only the leading characters that uniquely identify a property name. Case is ignored for property names.

`options = odeset(oldopts,'name1',value1,...)` alters an existing options structure `oldopts`. This sets `options` equal to the existing structure `oldopts`, overwrites any values in `oldopts` that are respecified using name/value pairs, and adds any new pairs to the structure. The modified structure is returned as an output argument.

`options = odeset(oldopts,newopts)` alters an existing options structure `oldopts` by combining it with a new options structure `newopts`. Any new options not equal to the empty matrix overwrite corresponding options in `oldopts`.

`odeset` with no input arguments displays all property names as well as their possible and default values.

ODE Properties

The following sections describe the properties that you can set using `odeset`. The available properties depend on the ODE solver you are using. There are several categories of properties:

- "Error Control Properties" on page 1-3750
- "Solver Output Properties" on page 1-3752
- "Step-Size Properties" on page 1-3755
- "Event Location Property" on page 1-3757
- "Jacobian Matrix Properties" on page 1-3758
- "Mass Matrix and DAE Properties" on page 1-3762
- "ode15s and ode15i-Specific Properties" on page 1-3764

Note This reference page describes the ODE properties for MATLAB, Version 7. The Version 5 properties are supported only for backward compatibility. For information on the Version 5 properties, type at the MATLAB command line: `more on, type odeset, more off`.

Error Control Properties

At each step, the solver estimates the local error `e` in the `i`th component of the solution. This error must be less than or equal to the acceptable error, which is a function of the specified relative tolerance, `RelTol`, and the specified absolute tolerance, `AbsTol`.

```
|e(i)| ≤ max(RelTol*abs(y(i)),AbsTol(i))
```

For routine problems, the ODE solvers deliver accuracy roughly equivalent to the accuracy you request. They deliver less accuracy for problems integrated over "long" intervals and problems that are moderately unstable. Difficult problems may require tighter tolerances than the default values. For relative accuracy, adjust `RelTol`. For

the absolute error tolerance, the scaling of the solution components is important: if |y| is somewhat smaller than AbsTol, the solver is not constrained to obtain any correct digits in y. You might have to solve a problem more than once to discover the scale of solution components.

Roughly speaking, this means that you want RelTol correct digits in all solution components except those smaller than thresholds AbsTol(i). Even if you are not interested in a component y(i) when it is small, you may have to specify AbsTol(i) small enough to get some correct digits in y(i) so that you can accurately compute more interesting components.

The following table describes the error control properties. Further information on each property is given following the table.

Property	Value	Description
RelTol	Positive scalar {1e-3}	Relative error tolerance that applies to all components of the solution vector y.
AbsTol	Positive scalar or vector {1e-6}	Absolute error tolerances that apply to the individual components of the solution vector.
NormControl	on | {off}	Control error relative to norm of solution.

Description of Error Control Properties

RelTol — This tolerance is a measure of the error relative to the size of each solution component. Roughly, it controls the number of correct digits in all solution components, except those smaller than thresholds AbsTol(i).

The default, 1e-3, corresponds to 0.1% accuracy.

AbsTol — AbsTol(i) is a threshold below which the value of the ith solution component is unimportant. The absolute error tolerances determine the accuracy when the solution approaches zero.

If AbsTol is a vector, the length of AbsTol must be the same as the length of the solution vector y. If AbsTol is a scalar, the value applies to all components of y.

NormControl — Set this property on to request that the solvers control the error in each integration step with norm(e) <= max(RelTol*norm(y),AbsTol). By default the solvers use a more stringent componentwise error control.

Solver Output Properties

The following table lists the solver output properties that control the output that the solvers generate. Further information on each property is given following the table.

Property	Value	Description
NonNegative	Vector of integers	Specifies which components of the solution vector must be nonnegative. The default value is [].
OutputFcn	Function handle	A function for the solver to call after every successful integration step.
OutputSel	Vector of indices	Specifies which components of the solution vector are to be passed to the output function.
Refine	Positive integer	Increases the number of output points by a factor of Refine.
Stats	on | {off}	Determines whether the solver should display statistics about its computations. By default, Stats is off.

Description of Solver Output Properties

NonNegative — The NonNegative property is not available in ode23s, ode15i. In ode15s, ode23t, and ode23tb, NonNegative is not available for problems where there is a mass matrix.

OutputFcn — To specify an output function, set 'OutputFcn' to a function handle. For example,

```
options = odeset('OutputFcn',@myfun)
```

sets 'OutputFcn' to @myfun, a handle to the function myfun. See the function_handle reference page for more information.

The output function must be of the form

```
status = myfun(t,y,flag)
```

"Parameterizing Functions" explains how to provide additional parameters to myfun, if necessary.

The solver calls the specified output function with the following flags. Note that the syntax of the call differs with the flag. The function must respond appropriately:

Flag	Description
init	The solver calls myfun(tspan,y0,'init') before beginning the integration to allow the output function to initialize. tspan and y0 are the input arguments to the ODE solver.
{[]}	The solver calls status = myfun(t,y,[]) after each integration step on which output is requested. t contains points where output was generated during the step, and y is the numerical solution at the points in t. If t is a vector, the ith column of y corresponds to the ith element of t. When length(tspan) > 2 the output is produced at every point in tspan. When length(tspan) = 2 the output is produced according to the Refine option.

Flag	Description
	`myfun` must return a `status` output value of `0` or `1`. If `status = 1`, the solver halts integration. You can use this mechanism, for instance, to implement a **Stop** button.
`done`	The solver calls `myfun([],[],'done')` when integration is complete to allow the output function to perform any cleanup chores.

You can use these general purpose output functions or you can edit them to create your own. Type `help function` at the command line for more information.

- `odeplot` — Time series plotting (default when you call the solver with no output arguments and you have not specified an output function)
- `odephas2` — Two-dimensional phase plane plotting
- `odephas3` — Three-dimensional phase plane plotting
- `odeprint` — Print solution as it is computed

Note If you call the solver with no output arguments, the solver does not allocate storage to hold the entire solution history.

`OutputSel` — Use `OutputSel` to specify which components of the solution vector you want passed to the output function. For example, if you want to use the `odeplot` output function, but you want to plot only the first and third components of the solution, you can do this using

```
options = ...
odeset('OutputFcn',@odeplot,'OutputSel',[1 3]);
```

By default, the solver passes all components of the solution to the output function.

`Refine` — If `Refine` is `1`, the solver returns solutions only at the end of each time step. If `Refine` is `n >1`, the solver subdivides each time step into `n` smaller intervals and returns solutions at each time point. `Refine` does not apply when `length(tspan)>2`.

Note In all the solvers, the default value of `Refine` is `1`. Within `ode45`, however, the default is `4` to compensate for the solver's large step sizes. To override this and see only the time steps chosen by `ode45`, set `Refine` to `1`.

The extra values produced for `Refine` are computed by means of continuous extension formulas. These are specialized formulas used by the ODE solvers to obtain accurate solutions between computed time steps without significant increase in computation time.

`Stats` — By default, `Stats` is `off`. If it is `on`, after solving the problem the solver displays

- Number of successful steps
- Number of failed attempts
- Number of times the ODE function was called to evaluate $f(t,y)$

Solvers based on implicit methods, including `ode23s`, `ode23t`, `ode15s`, and `ode15i`, also display

- Number of times that the partial derivatives matrix $\partial f/\partial x$ was formed
- Number of LU decompositions
- Number of solutions of linear systems

Step-Size Properties

The step-size properties specify the size of the first step the solver tries, potentially helping it to better recognize the scale of the problem. In addition, you can specify bounds on the sizes of subsequent time steps.

The following table describes the step-size properties. Further information on each property is given following the table.

Property	Value	Description
`InitialStep`	Positive scalar	Suggested initial step size.
`MaxStep`	Positive scalar {`0.1*abs(t0-tf)`}	Upper bound on solver step size.

Description of Step-Size Properties

`InitialStep` — `InitialStep` sets an upper bound on the magnitude of the first step size the solver tries. If you do not set `InitialStep`, the initial step size is based on the slope of the solution at the initial time `tspan(1)`, and if the slope of all solution components is zero, the procedure might try a step size that is much too large. If you know this is happening or you want to be sure that the solver resolves important behavior at the start of the integration, help the code start by providing a suitable `InitialStep`.

`MaxStep` — If the differential equation has periodic coefficients or solutions, it might be a good idea to set `MaxStep` to some fraction (such as 1/4) of the period. This guarantees that the solver does not enlarge the time step too much and step over a period of interest. Do *not* reduce `MaxStep` for any of the following purposes:

- To produce more output points. This can significantly slow down solution time. Instead, use `Refine` to compute additional outputs by continuous extension at very low cost.
- When the solution does not appear to be accurate enough. Instead, reduce the relative error tolerance `RelTol`, and use the solution you just computed to determine appropriate values for the absolute error tolerance vector `AbsTol`. See “Error Control Properties” on page 1-3750 for a description of the error tolerance properties.
- To make sure that the solver doesn’t step over some behavior that occurs only once during the simulation interval. If you know the time at which the change occurs, break the simulation interval into two pieces and call the solver twice. If you do not know the time at which the change occurs, try reducing the error tolerances `RelTol` and `AbsTol`. Use `MaxStep` as a last resort.

Event Location Property

In some ODE problems the times of specific events are important, such as the time at which a ball hits the ground, or the time at which a spaceship returns to the earth. While solving a problem, the ODE solvers can detect such events by locating transitions to, from, or through zeros of user-defined functions.

The following table describes the `Events` property. Further information on each property is given following the table.

ODE Events Property

String	Value	Description
`Events`	Function handle	Handle to a function that includes one or more event functions.

Description of Event Location Properties

`Events` — The function is of the form

```
[value,isterminal,direction] = events(t,y)
```

`value`, `isterminal`, and `direction` are vectors for which the `i`th element corresponds to the `i`th event function:

- `value(i)` is the value of the `i`th event function.
- `isterminal(i)` = `1` if the integration is to terminate at a zero of this event function, otherwise, `0`.
- `direction(i) = 0` if all zeros are to be located (the default), `+1` if only zeros where the event function is increasing, and `-1` if only zeros where the event function is decreasing.

If you specify an events function and events are detected, the solver returns three additional outputs:

- `A` column vector of times at which events occur
- Solution values corresponding to these times

- Indices into the vector returned by the events function. The values indicate which event the solver detected.

If you call the solver as

```
[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)
```

the solver returns these outputs as `TE`, `YE`, and `IE` respectively. If you call the solver as

```
sol = solver(odefun,tspan,y0,options)
```

the solver returns these outputs as `sol.xe`, `sol.ye`, and `sol.ie`, respectively.

For examples that use an event function, see "Event Location" and "Advanced Event Location" in the MATLAB Mathematics documentation.

Jacobian Matrix Properties

The stiff ODE solvers often execute faster if you provide additional information about the Jacobian matrix $\partial f/\partial y$, a matrix of partial derivatives of the function that defines the differential equations.

$$\frac{\partial f}{\partial y} = \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} & \cdots \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} & \cdots \\ \vdots & \vdots & \end{bmatrix}$$

The Jacobian matrix properties pertain only to those solvers for stiff problems (`ode15s`, `ode23s`, `ode23t`, `ode23tb`, and `ode15i`) for which the Jacobian matrix $\partial f/\partial y$ can be critical to reliability and efficiency. If you do not provide a function to calculate the Jacobian, these solvers approximate the Jacobian numerically using finite differences. In this case, you might want to use the `Vectorized` or `JPattern` properties.

The following table describes the Jacobian matrix properties for all implicit solvers except `ode15i`. Further information on each property is given following the table. See Jacobian Properties for ode15i on page 1-3761 for `ode15i`-specific information.

Jacobian Properties for All Implicit Solvers Except ode15i

Property	Value	Description
`Jacobian`	Function handle | constant matrix	Matrix or function that evaluates the Jacobian.
`JPattern`	Sparse matrix of {0,1}	Generates a sparse Jacobian matrix numerically.
`Vectorized`	`on` | {`off`}	Allows the solver to reduce the number of function evaluations required.

Description of Jacobian Properties

`Jacobian` — Supplying an analytical Jacobian often increases the speed and reliability of the solution for stiff problems. Set this property to a function `FJac`, where `FJac(t,y)` computes $\partial f/\partial y$, or to the constant value of $\partial f/\partial y$.

The Jacobian for the "van der Pol Equation (Stiff)", described in the MATLAB Mathematics documentation, can be coded as

```
function J = vdp1000jac(t,y)
J = [  0                   1
     (-2000*y(1)*y(2)-1)  (1000*(1-y(1)^2)) ];
```

`JPattern` — `JPattern` is a sparsity pattern with `1`s where there might be nonzero entries in the Jacobian.

Note If you specify `Jacobian`, the solver ignores any setting for `JPattern`.

Set this property to a sparse matrix S with $S(i,j) = 1$ if component i of $f(t,y)$ depends on component j of y, and 0 otherwise. The solver uses this sparsity pattern to generate a sparse Jacobian matrix numerically. If the Jacobian matrix is large and sparse, this can greatly accelerate execution. For an example using the `JPattern` property, see Example: Large, Stiff, Sparse Problem in the MATLAB Mathematics documentation.

`Vectorized` — The `Vectorized` property allows the solver to reduce the number of function evaluations required to compute all the columns of the Jacobian matrix, and might significantly reduce solution time.

Set `on` to inform the solver that you have coded the ODE function `F` so that `F(t,[y1 y2 ...])` returns `[F(t,y1) F(t,y2) ...]`. This allows the solver to reduce the number of function evaluations required to compute all the columns of the Jacobian matrix, and might significantly reduce solution time.

Note If you specify `Jacobian`, the solver ignores a setting of `'on'` for `'Vectorized'`.

With the MATLAB array notation, it is typically an easy matter to vectorize an ODE function. For example, you can vectorize the "van der Pol Equation (Stiff)", described in the MATLAB Mathematics documentation, by introducing colon notation into the subscripts and by using the array power (`.^`) and array multiplication (`.*`) operators.

```
function dydt = vdp1000(t,y)
dydt = [y(2,:); 1000*(1-y(1,:).^2).*y(2,:)-y(1,:)];
```

Note Vectorization of the ODE function used by the ODE solvers differs from the vectorization used by the boundary value problem (BVP) solver, `bvp4c`. For the ODE solvers, the ODE function is vectorized only with respect to the second argument, while `bvp4c` requires vectorization with respect to the first and second arguments.

The following table describes the Jacobian matrix properties for `ode15i`.

Jacobian Properties for ode15i

Property	Value	Description
`Jacobian`	Function handle | Cell array of constant values	Function that evaluates the Jacobian or a cell array of constant values.
`JPattern`	Sparse matrices of {0,1}	Generates a sparse Jacobian matrix numerically.
`Vectorized`	`on` | {`off`}	Vectorized ODE function

Description of Jacobian Properties for ode15i

`Jacobian` — Supplying an analytical Jacobian often increases the speed and reliability of the solution for stiff problems. Set this property to a function

```
[dFdy,dFdp] = Fjac(t,y,yp)
```

or to a cell array of constant values $[\partial F/\partial y, (\partial F/\partial y)']$.

`JPattern` — `JPattern` is a sparsity pattern with `1`'s where there might be nonzero entries in the Jacobian.

Set this property to `{dFdyPattern,dFdypPattern}`, the sparsity patterns of $\partial F/\partial y$ and $\partial F/\partial y'$, respectively.

`Vectorized` —

Set this property to `{yVect, ypVect}`. Setting `yVect` to `'on'` indicates that

```
F(t, [y1 y2 ...], yp)
```

returns

```
[F(t,y1,yp), F(t,y2,yp) ...]
```

Setting `ypVect` to `'on'` indicates that

```
F(t,y,[yp1 yp2 ...])
```

returns

```
[F(t,y,yp1) F(t,y,yp2) ...]
```

Mass Matrix and DAE Properties

This section describes mass matrix and differential-algebraic equation (DAE) properties, which apply to all the solvers except `ode15i`. These properties are not applicable to `ode15i` and their settings do not affect its behavior.

The solvers of the ODE suite can solve ODEs of the form

$$M(t,y)y' = f(t,y) \tag{1-3}$$

with a mass matrix $M(t,y)$ that can be sparse.

When $M(t,y)$ is nonsingular, the equation above is equivalent to $y' = M^{-1} f(t,y)$ and the ODE has a solution for any initial values y_0 at t_0. The more general form (Equation 1-3) is convenient when you express a model naturally in terms of a mass matrix. For large, sparse $M(t,y)$, solving Equation 1-3 directly reduces the storage and run-time needed to solve the problem.

When $M(t,y)$ is singular, then $M(t,y)$ times $M(t,y)y' = f(t,y)$ is a DAE. A DAE has a solution only when y_0 is consistent; that is, there exists an initial slope yp_0 such that $M(t_0,y_0)yp_0 = f(t_0,y_0)$. If y_0 and yp_0 are not consistent, the solver treats them as guesses, attempts to compute consistent values that are close to the guesses, and continues to solve

the problem. For DAEs of index 1, solving an initial value problem with consistent initial conditions is much like solving an ODE.

The `ode15s` and `ode23t` solvers can solve DAEs of index 1. For examples of DAE problems, see Example: Differential-Algebraic Problem, in the MATLAB Mathematics documentation, and the examples `amp1dae` and `hb1dae`.

The following table describes the mass matrix and DAE properties. Further information on each property is given following the table.

Mass Matrix and DAE Properties (Solvers Other Than ode15i)

Property	Value	Description
`Mass`	Matrix | function handle	Mass matrix or a function that evaluates the mass matrix $M(t,y)$.
`MStateDependence`	`none	{weak}	strong`	Dependence of the mass matrix on y.
`MvPattern`	Sparse matrix	$\partial(M(t,y)v)/\partial y$ sparsity pattern.
`MassSingular`	`yes	no	{maybe}`	Indicates whether the mass matrix is singular.
`InitialSlope`	Vector {zero vector}	Vector representing the consistent initial slope yp_0.

Description of Mass Matrix and DAE Properties

`Mass` — For problems of the form $M(t)y' = f(t,y)$, set `'Mass'` to a mass matrix M. For problems of the form $M(t)y' = f(t,y)$, set `'Mass'` to a function handle `@Mfun`, where `Mfun(t,y)` evaluates the mass matrix $M(t,y)$. The `ode23s` solver can only solve problems with a constant mass matrix M. When solving DAEs, using `ode15s` or `ode23t`, it is advantageous to formulate the problem so that M is a diagonal matrix (a semiexplicit DAE).

For example problems, see "Finite Element Discretization" in the MATLAB Mathematics documentation, or the examples `fem2ode` or `batonode`.

`MStateDependence` — Set this property to `none` for problems $M(t)y' = f(t, y)$. Both `weak` and `strong` indicate $M(t,y)$, but `weak` results in implicit solvers using approximations when solving algebraic equations.

`MvPattern` — Set this property to a sparse matrix S with $S(i,j) = 1$ if, for any k, the (i,k) component of $M(t,y)$ depends on component j of y, and 0 otherwise. For use with the `ode15s`, `ode23t`, and `ode23tb` solvers when `MStateDependence` is `strong`. See `burgersode` as an example.

`MassSingular` — Set this property to `no` if the mass matrix is not singular and you are using either the `ode15s` or `ode23t` solver. The default value of `maybe` causes the solver to test whether the problem is a DAE, by testing whether $M(t_0,y_0)$ is singular.

`InitialSlope` — Vector representing the consistent initial slope yp_0, where yp_0 satisfies $M(t_0, y_0) \cdot yp_0 = f(t_0, y_0)$. The default is the zero vector.

This property is for use with the `ode15s` and `ode23t` solvers when solving DAEs.

ode15s and ode15i-Specific Properties

`ode15s` is a variable-order solver for stiff problems. It is based on the numerical differentiation formulas (NDFs). The NDFs are generally more efficient than the closely related family of backward differentiation formulas (BDFs), also known as Gear's methods. The `ode15s` properties let you choose among these formulas, as well as specifying the maximum order for the formula used.

`ode15i` solves fully implicit differential equations of the form

$$f(t, y, y') = 0$$

using the variable order BDF method.

The following table describes the ode15s and ode15i-specific properties. Further information on each property is given following the table. Use odeset to set these properties.

ode15s and ode15i-Specific Properties

Property	Value	Description
MaxOrder	1 | 2 | 3 | 4 | {5}	Maximum order formula used to compute the solution.
BDF (ode15s only)	on | {off}	Specifies whether you want to use the BDFs instead of the default NDFs.

Description of ode15s and ode15i-Specific Properties

MaxOrder — Maximum order formula used to compute the solution.

BDF (ode15s only) — Set BDF on to have ode15s use the BDFs.

For both the NDFs and BDFs, the formulas of orders 1 and 2 are A-stable (the stability region includes the entire left half complex plane). The higher order formulas are not as stable, and the higher the order the worse the stability. There is a class of stiff problems (stiff oscillatory) that is solved more efficiently if MaxOrder is reduced (for example to 2) so that only the most stable formulas are used.

See Also

deval | odeget | ode45 | ode23 | ode23t | ode23tb | ode113 | ode15s | ode23s | function_handle

odextend

Purpose

Extend solution of initial value problem for ordinary differential equation

Syntax

```
solext = odextend(sol, odefun, tfinal)
solext = odextend(sol,[],tfinal)
solext = odextend(sol, odefun, tfinal, yinit)
solext = odextend(sol, odefun, tfinal, [yinit, ypinit])
solext = odextend(sol, odefun, tfinal, yinit, options)
```

Description

`solext = odextend(sol, odefun, tfinal)` extends the solution stored in `sol` to an interval with upper bound `tfinal` for the independent variable. Specify `odefun` as a function handle. Specify `sol` as an ODE solution structure created using an ODE solver. The lower bound for the independent variable in `solext` is the same as in `sol`. If you created `sol` with an ODE solver other than `ode15i`, the function `odefun` computes the right-hand side of the ODE equation, which is of the form $y' = f(t,y)$. If you created `sol` using `ode15i`, the function `odefun` computes the left-hand side of the ODE equation, which is of the form $f(t,y,y') = 0$.

"Parameterizing Functions" explains how to provide additional parameters to the function `odefun`, if necessary.

`odextend` extends the solution by integrating `odefun` from the upper bound for the independent variable in `sol` to `tfinal`, using the same ODE solver that created `sol`. By default, `odextend` uses

- The initial conditions `y = sol.y(:,end)` for the subsequent integration
- The same integration properties and additional input arguments the ODE solver originally used to compute `sol`. This information is stored as part of the solution structure `sol` and is subsequently passed to `solext`. Unless you want to change these values, you do not need to pass them to `odextend`.

`solext = odextend(sol,[],tfinal)` uses the same ODE function that the ODE solver uses to compute `sol` to extend the solution. It is

not necessary to pass in `odefun` explicitly unless it differs from the original ODE function.

`solext = odextend(sol, odefun, tfinal, yinit)` uses the column vector `yinit` as new initial conditions for the subsequent integration, instead of the vector `sol.y(end)`.

Note To extend solutions obtained with `ode15i`, use the following syntax, in which the column vector `ypinit` is the initial derivative of the solution:

```
solext = odextend(sol, odefun, tfinal, [yinit, ypinit])
```

`solext = odextend(sol, odefun, tfinal, yinit, options)` uses the integration properties specified in `options` instead of the options the ODE solver originally used to compute `sol`. The new options are then stored within the structure `solext`. See `odeset` for details on setting `options` properties. Set `yinit = []` as a placeholder to specify the default initial conditions.

Examples

The following command

```
sol=ode45(@vdp1,[0 10],[2 0]);
```

uses `ode45` to solve the system `y' = vdp1(t,y)`, where `vdp1` is an example of an ODE function provided with MATLAB software, on the interval `[0 10]`. Then, the commands

```
sol=odextend(sol,@vdp1,20);
plot(sol.x,sol.y(1,:));
```

extend the solution to the interval `[0 20]` and plot the first component of the solution on `[0 20]`.

See Also

`deval` | `ode23` | `ode45` | `ode113` | `ode15s` | `ode23s` | `ode23t` | `ode23tb` | `ode15i` | `odeset` | `odeget` | `deval` | `function_handle`

onCleanup

Purpose

Cleanup tasks upon function completion

Syntax

`cleanupObj = onCleanup(cleanupFun)`

Description

`cleanupObj = onCleanup(cleanupFun)` creates an object that, when destroyed, executes the function `cleanupFun`. MATLAB implicitly clears all local variables at the termination of a function, whether by normal completion, or a forced exit, such as an error, or **Ctrl+C**.

If you reference or pass `cleanupObj` outside your function, then `cleanupFun` does not run when that function terminates. Instead, it runs whenever the MATLAB destroys the object.

Input Arguments

cleanupFun - Clean-up task

function handle

Clean-up task, specified as a function handle. Your clean-up function should never rely on variables that are defined outside of the scope of that function.

You can declare any number of `onCleanup` objects in a program file. However, if the clean-up tasks depend on the order of execution, then you should define only one object that calls a script or function, containing the relevant clean-up commands.

You should use an anonymous function handle to call your clean-up task. This allows you to pass arguments to your clean-up function.

Example: `@() fclose(file.m)`

Example: `@() user_script`

Example: `@() function(input)`

Data Types
`function_handle`

Examples

Close a File After Executing Function

Save the following code in `action.m` and type `action` in the Command Window.

```
 function [] = action()
    file = fullfile(matlabroot,'help','techdoc','matlab_env',...
     'examples','collatz.m');
    fid = fopen(file);
    finishup = onCleanup(@() fclose(fid));
    disp('Execute code');
    disp('then close file')
 end
```

```
Execute code
then close file
```

Switch Directories After Executing Function

Pass your own script to the `onCleanup` object so that it executes MATLAB destroys the cleanup object.

Save the following code in `cleanup.m`.

```
cd(matlabroot);
disp('You are now in the MATLAB root directory')
```

Save the following code in `youraction.m` and type `youraction` in the Command Window.

```
function [] = youraction
 changeup = onCleanup(@() cleanup);
 disp('Execute Code');
end
```

```
Execute Code
You are now in the MATLAB root directory
```

Tips

- If your program contains multiple cleanup objects, MATLAB does not guarantee the order that it destroys these objects.
- If the order of your cleanup functions matters, define one `onCleanup` object for all the tasks.

See Also

`clear` | `clearvars` | `function_handle`

Concepts

- “Clean Up When Functions Complete”
- “Object Lifecycle”
- “Function Handles”
- “What Are Anonymous Functions?”

Purpose

Create array of all ones

Syntax

```
X = ones
X = ones(n)
X = ones(sz1,...,szN)
X = ones(sz)

X = ones(classname)
X = ones(n,classname)
X = ones(sz1,...,szN,classname)
X = ones(sz,classname)

X = ones('like',p)
X = ones(n,'like',p)
X = ones(sz1,...,szN,'like',p)
X = ones(sz,'like',p)
```

Description

`X = ones` returns the scalar `1`.

`X = ones(n)` returns an `n`-by-`n` matrix of ones.

`X = ones(sz1,...,szN)` returns an `sz1`-by-...-by-`szN` array of ones where `sz1,...,szN` indicates the size of each dimension. For example, `ones(2,3)` returns a 2-by-3 array of ones.

`X = ones(sz)` returns an array of ones where the size vector, `sz`, defines `size(X)`. For example, `ones([2,3])` returns a 2-by-3 array of ones.

`X = ones(classname)` returns a scalar `1` where the string, `classname`, specifies the data type. For example, `ones('int8')` returns a scalar, 8-bit integer `1`.

`X = ones(n,classname)` returns an `n`-by-`n` array of ones of data type `classname`.

`X = ones(sz1,...,szN,classname)` returns an `sz1`-by-...-by-`szN` array of ones of data type `classname`.

`X = ones(sz,classname)` returns an array of ones where the size vector, `sz`, defines `size(X)` and `classname` defines `class(X)`.

`X = ones('like',p)`returns a scalar `1` with the same data type, sparsity, and complexity (real or complex) as the numeric variable, `p`.

`X = ones(n,'like',p)` returns an `n`-by-`n` array of ones like `p`.

`X = ones(sz1,...,szN,'like',p)` returns an `sz1`-by-...-by-`szN` array of ones like `p`.

`X = ones(sz,'like',p)` returns an array of ones like `p` where the size vector, `sz`, defines `size(X)`.

Input Arguments

n - Size of square matrix

integer value

Size of square matrix, specified as an integer value, defines the output as a square, n-by-n matrix of ones.

- If `n` is `0`, then `X` is an empty matrix.
- If `n` is negative, then it is treated as `0`.

Data Types

`double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

sz1,...,szN - Size of each dimension

two or more integer values

Size of each dimension, specified as two or more integer values, defines `X` as a sz1-by...-by-szN array.

- If the size of any dimension is `0`, then `X` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- If any trailing dimensions greater than `2` have a size of `1`, then the output, `X`, does not include those dimensions.

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

sz - Output size

row vector of integer values

Output size, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension.

- If the size of any dimension is `0`, then `X` is an empty array.
- If the size of any dimension is negative, then it is treated as `0`.
- If any trailing dimensions greater than `2` have a size of `1`, then the output, `X`, does not include those dimensions.

Example: `sz = [2,3,4]` defines `X` as a 2-by-3-by-4 array.

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

classname - Output class

`'double'` (default) | `'single'` | `'int8'` | `'uint8'`

Output class, specified as `'double'`, `'single'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'int64'`, or `'uint64'`.

Data Types
`char`

p - Prototype

numeric variable

Prototype, specified as a numeric variable.

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

Examples

Square Array of Ones

Create a 4-by-4 array of ones.

```
X = ones(4)
```

```
X =

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
```

3-D Array of Ones

Create a 2-by-3-by-4 array of ones.

```
X = ones(2,3,4);

size(X)
```

```
ans =

     2     3     4
```

Size Defined by Existing Array

Define a 3-by-2 array A.

```
A = [1 4 ; 2 5 ; 3 6];

sz = size(A)
```

```
sz =

     3     2
```

Create an array of ones that is the same size as A

```
X = ones(sz)
```

```
X =

     1     1
     1     1
     1     1
```

Nondefault Numeric Data Type

Create a 1-by-3 vector of ones whose elements are 16-bit unsigned integers.

```
X = ones(1,3,'uint16'),
class(X)
```

```
X =

      1      1      1


ans =

uint16
```

Complex One

Create a scalar 1 that is not real valued, but instead is complex like an existing array.

Define a complex vector.

```
p = [1+2i 3i];
```

Create a scalar 1 that is complex like p.

```
X = ones('like',p)
```

```
X =

   1.0000 + 0.0000i
```

Size and Numeric Data Type of Defined by Existing Array

Define a 2-by-3 array of 8-bit unsigned integers.

```
p = uint8([1 3 5 ; 2 4 6]);
```

Create an array of ones that is the same size and data type as p.

```
X = ones(size(p),'like',p),
class(X)
```

```
X =

    1    1    1
    1    1    1


ans =

uint8
```

See Also

eye | ones | rand | randn | complex | false | size

Concepts

- “Preallocating Arrays”

Purpose

Open file in appropriate application

Syntax

```
open(name)
output = open(name)
```

Description

`open(name)` opens the specified file or variable in the appropriate application.

`output = open(name)` returns an empty `output` (`[]`) for most cases. If opening a MAT-file, `output` is a structure that contains the variables in the file. If opening a figure, `output` is a handle to that figure.

Tips

The `open` function opens files based on their extension. You can extend the functionality of `open` by defining your own file handling function of the form `openxxx`, where *xxx* is a file extension. For example, if you create a function `openlog`, the `open` function calls `openlog` to process any files with the `.log` extension. The `open` function returns any single *output* defined by your function.

Input Arguments

name

Name of file or variable to open. If *name* does not include an extension, the `open` function:

1. Searches for a variable named *name*. If the variable exists, `open` opens it in the Variables editor.

2. Searches the MATLAB path for *name*`.mdl`, *name*`.slx`, or *name*`.m`. If *name*`.mdl` or *name*`.slx` exists, then `open` opens the model in Simulink. If only *name*`.m` exists, `open` opens the file in the MATLAB Editor.

If more than one file named *name* exists on the MATLAB path, the `open` function opens the file returned by `which(name)`.

The `open` function performs the following actions based on the file extension:

(Continued)

`.m`	Open in MATLAB Editor.
`.mat`	Return variables in structure *st* when called with the syntax: *st* = open(*name*)
`.fig`	Open figure in Handle Graphics.
`.mdl` or `.slx`	Open model in Simulink.
`.prj`	Open project in the MATLAB Compiler Deployment Tool.
`.doc*`	Open document in Microsoft Word.
`.exe`	Run executable file (only on Windows systems).
`.pdf`	Open document in Adobe® Acrobat®.
`.ppt*`	Open document in Microsoft PowerPoint.
`.xls*`	Start MATLAB Import Wizard.
`.htm` or `.html`	Open document in MATLAB browser.
`.url`	Open file in your default Web browser.

Examples

Open `Contents.m` in the MATLAB Editor by typing:

```
open Contents.m
```

Generally, MATLAB opens *matlabroot*`\toolbox\matlab\general\Contents.m`. However, if you have a file called `Contents.m` in a directory that is before `toolbox\matlab\general` on the MATLAB path, then `open` opens that file instead.

Open a file not on the MATLAB path by including the complete file specification:

```
open('D:\temp\data.mat')
```

If the file does not exist, MATLAB displays an error message.

Create a function called `opentxt` to handle files with extension `.txt`:

```
function opentxt(filename)

   fprintf('You have requested file: %s\n', filename);

   wh = which(filename);
   if exist(filename, 'file') == 2
     fprintf('Opening in MATLAB Editor: %s\n', filename);
     edit(filename);
   elseif ~isempty(wh)
     fprintf('Opening in MATLAB Editor: %s\n', wh);
     edit(wh);
   else
     warning('MATLAB:fileNotFound', ...
             'File was not found: %s', filename);
   end

end
```

Open the file `ngc6543a.txt` (a description of `ngc6543a.jpg`, located in *`matlabroot`*`\toolbox\matlab\demos`):

```
photo_text = 'ngc6543a.txt';
open(photo_text)
```

`open` calls your function with the following syntax:

```
opentxt(photo_text)
```

See Also

`edit` | `load` | `openfig` | `openvar` | `path` | `uiopen` | `which` | `winopen`

VideoWriter.open

Purpose

Open file for writing video data

Syntax

```
open(writerObj)
```

Description

`open(writerObj)` opens the file associated with `writerObj` for writing. When you open the file, all properties of the object become read only. `open` discards any existing contents of the file.

Input Arguments

writerObj

`VideoWriter` object created by the `VideoWriter` function.

Examples

Open a new AVI file:

```
myObj = VideoWriter('newfile.avi');
open(myObj);
```

See Also

`writeVideo` | `VideoWriter` | `close`

Purpose

Open new copy or raise existing copy of saved figure

Syntax

```
openfig('filename.fig')
openfig('filename.fig','new')
openfig('filename.fig','reuse')
openfig('filename.fig','new','invisible')
openfig('filename.fig','reuse','invisible')
openfig('filename.fig','new','visible')
openfig('filename.fig','reuse','visible')
figure_handle = openfig(...)
```

Description

`openfig('filename.fig')` and `openfig('filename.fig','new')` opens the figure contained in the FIG-file, `filename.fig`, and ensures it is visible and positioned completely on screen. You do not have to specify the full path to the FIG-file as long as it is on your MATLAB path. The `.fig` extension is optional.

`openfig('filename.fig','reuse')` opens the figure contained in the FIG-file only if a copy of the figure is not currently open. Otherwise, `openfig` brings the existing copy forward, making sure it is still visible and completely on screen.

`openfig('filename.fig','new','invisible')` or `openfig('filename.fig','reuse','invisible')` opens the figure as in the preceding example, while forcing the figure to be invisible.

`openfig('filename.fig','new','visible')` or `openfig('filename.fig','reuse','visible')` opens the figure, while forcing the figure to be visible.

`figure_handle = openfig(...)` returns the handle to the figure.

Tips

`openfig` is designed for use with GUI figures. Use this function to:

- Open the FIG-file creating the GUI and ensure it is displayed on screen. This provides compatibility with different screen sizes and resolutions.

- Control whether the MATLAB software displays one or multiple instances of the GUI at any given time.
- Return the handle of the figure created, which is typically hidden for GUI figures.

If the FIG-file contains an invisible figure, `openfig` returns its handle and leaves it invisible. The caller should make the figure visible when appropriate.

Do not use `openfig` or double-click a FIG-file to open a GUI created with GUIDE. Instead open the GUI code file by typing its name in the command window or by right-clicking its name in the Current Folder Browser and selecting **Run File**. To open a GUIDE GUI, for example one called `guifile.m`, in an invisible state, specify the `Visible` property in your command:

```
guifile('Visible','off')
```

Your code should then make the figure visible at an appropriate time.

See Also

`guide` | `guihandles` | `movegui` | `open` | `hgload` | `save`

Purpose

Control OpenGL rendering

Syntax

```
opengl info
s = opengl('data')
opengl software
opengl hardware
opengl verbose
opengl quiet
opengl DriverBugWorkaround
opengl('DriverBugWorkaround',WorkaroundState)
```

Description

The OpenGL autoselection mode applies when the `RendererMode` of the figure is `auto`. Possible values for *`selection_mode`* are

- `autoselect` – allows OpenGL to be automatically selected if OpenGL is available and if there is graphics hardware on the host machine.
- `neverselect` – disables autoselection of OpenGL.
- `advise` – prints a message to the command window if OpenGL rendering is advised, but `RenderMode` is set to `manual`.

`opengl`, by itself, returns the current autoselection state.

Note that the autoselection state only specifies whether OpenGL should or should not be considered for rendering; it does not explicitly set the rendering to OpenGL. You can do this by setting the `Renderer` property of the figure to `OpenGL`. For example,

```
set(figure_handle,'Renderer','OpenGL')
```

`opengl info` prints information with the version and vendor of the OpenGL on your system. Also indicates wether your system is currently using hardware of software OpenGL and the state of various driver bug workarounds. Note that calling `opengl info` loads the OpenGL Library.

For example, the following output is generated on a Windows XP computer that uses ATI Technologies graphics hardware:

```
>> opengl info
Version         = 1.3.4010 WinXP Release
Vendor          = ATI Technologies Inc.
Renderer        = RADEON 9600SE x86/SSE2
MaxTextureSize  = 2048
Visual          = 05 (RGB 16 bits(05 06 05 00) zdepth 16, Hardware
Accelerated, Opengl, Double Buffered, Window)
Software        = false
# of Extensions = 85
Driver Bug Workarounds:
OpenGLBitmapZbufferBug    = 0
OpenGLWobbleTesselatorBug = 0
OpenGLLineSmoothingBug    = 0
OpenGLDockingBug          = 0
OpenGLClippedImageBug     = 0
```

Note that different computer systems may not list all OpenGL bugs.

`s = opengl('data')` returns a structure containing the same data that is displayed when you call `opengl info`, with the exception of the driver bug workaround state.

`opengl software` forces the MATLAB software to use software OpenGL rendering instead of hardware OpenGL. Note that Macintosh systems do not support software OpenGL.

`opengl hardware` reverses the `opengl software` command and enables MATLAB to use hardware OpenGL rendering if it is available. If your computer does not have OpenGL hardware acceleration, MATLAB automatically switches to software OpenGL rendering (except on Macintosh systems, which do not support software OpenGL).

Note that on UNIX systems, the `software` or `hardware` options with the `opengl` command works only if MATLAB has not yet used the OpenGL renderer or you have not issued the `opengl info` command (which attempts to load the OpenGL Library).

`opengl verbose` displays verbose messages about OpenGL initialization (if OpenGL is not already loaded) and other runtime messages.

`opengl quiet` disables verbose message setting.

`opengl` *`DriverBugWorkaround`* queries the state of the specified driver bug workaround. Use the command `opengl info` to see a list of all driver bug workarounds. See "Driver Bug Workarounds" on page 1-3785 for more information.

`opengl('`*`DriverBugWorkaround`*`',WorkaroundState)` sets the state of the specified driver bug workaround. You can set WorkaroundState to one of three values:

- `0` – Disable the specified *`DriverBugWorkaround`* (if enabled) and do not allow MATLAB to autoselect this workaround.
- `1` – Enable the specified *`DriverBugWorkaround`*.
- `-1` – Set the specified *`DriverBugWorkaround`* to autoselection mode, which allows MATLAB to enable this workaround if the requisite conditions exist.

Driver Bug Workarounds

The MATLAB software enables various OpenGL driver bug workarounds when it detects certain known problems with installed hardware. However, because there are many versions of graphics drivers, you might encounter situations when MATLAB does not enable a workaround that would solve a problem you are having with OpenGL rendering.

This section describes the symptoms that each workaround is designed to correct so you can decide if you want to try using one to fix an OpenGL rendering problem.

Use the `opengl info` command to see what driver bug workarounds are available on your computer.

Note These workarounds have not been tested under all driver combinations and therefore might produce undesirable results under certain conditions.

OpenGLBitmapZbufferBug

Symptom: text with background color (including data tips) and text displayed on image, patch, or surface objects is not visible when using OpenGL renderer.

Possible side effect: text is always on top of other objects.

Command to enable:

```
opengl('OpenGLBitmapZbufferBug',1)
```

OpenGLWobbleTesselatorBug

Symptom: Rendering complex patch object causes segmentation violation and returns a tesselator error message in the stack trace.

Command to enable:

```
opengl('OpenGLWobbleTesselatorBug',1)
```

OpenGLLineSmoothingBug

Symptom: Lines with a `LineWidth` greater than 3 look bad.

Command to enable:

```
opengl('OpenGLLineSmoothingBug',1)
```

OpenGLDockingBug

Symptom: MATLAB crashes when you dock a figure that has its `Renderer` property set to `opengl`.

Command to enable:

```
opengl('OpenGLDockingBug',1)
```

OpenGLClippedImageBug

Symptom: Images (as well as `colorbar` displays) do not display when the `Renderer` property set to `opengl`.

Command to enable:

```
opengl('OpenGLClippedImageBug',1)
```

OpenGLEraseModeBug

Symptom: Graphics objects with `EraseMode` property set to non-normal erase modes (`xor`, `none`, or `background`) do not draw when the figure `Renderer` property is set to `opengl`.

Command to enable:

```
opengl('OpenGLEraseModeBug',1)
```

See Also

`Renderer`

openvar

Purpose

Open workspace variable in Variables editor or other graphical editing tool

Syntax

```
openvar(varname)
```

Description

`openvar(varname)` opens the workspace variable named by the string, `varname`, in the Variables editor for graphical editing. Changes that you make to variables in the Variables editor occur in the workspace as soon as you enter them.

In some toolboxes, `openvar` opens a tool appropriate for viewing or editing objects indicated by `varname` instead of opening the Variables editor.

MATLAB does not impose any limitation on the size of a variable that you can open in the Variables editor. However, your operating system or the amount of physical memory installed on your computer can impose such limits.

Input Arguments

varname - Variable name

string

Variable name, specified as a string. The named variable can be an array, character string, cell array, structure, or an object and its properties. If the named variable is a multidimensional array, then you can only view the array in the Variables editor, and not edit it.

Example: `'myVariable'`

Example: `'A'`

Examples

Identify Outliers in a Linked Graph

Use data brushing to identify observations in a vector or matrix that might warrant further analysis.

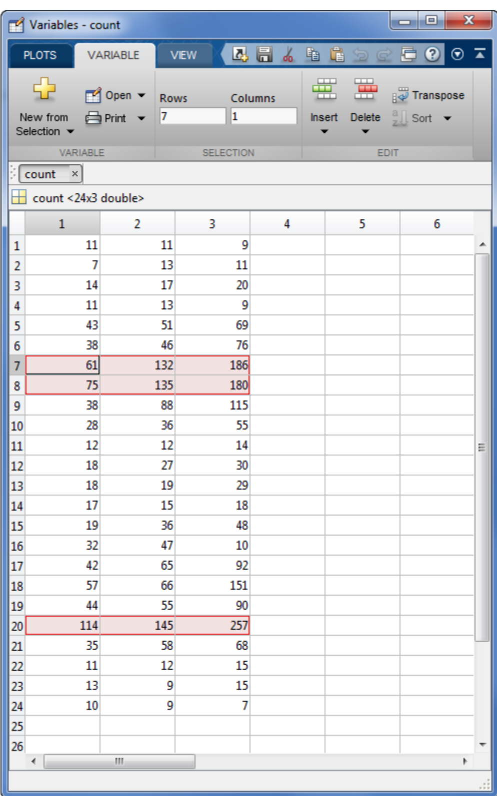
Make a scatter plot of data in the sample MAT-file `count.dat`, and open the variable `count` in the Variables editor.

```
load count.dat
```

```
scatter(count(:,1),count(:,2))
openvar('count')
```

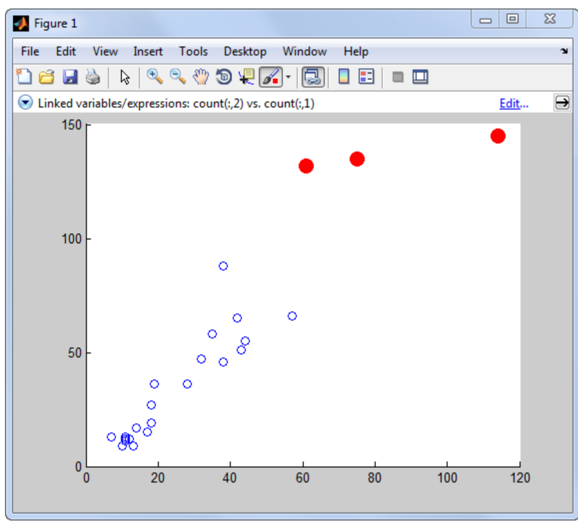
Right-click a cell in the Variables editor and select **Brushing > Brushing On**. This turns on data brushing in the Variables editor.

Select the rows 7, 8, and 20. (Select noncontiguous rows by holding down the **Ctrl** key and clicking in each row.)



In the Figure window with the scatter plot, click **Brush/Select Data** to enable data brushing, and **Link Plot** to enable data linking.

The data observations you brushed in the Variables editor appear highlighted in the scatter plot.



As long as data linking is enabled in the figure, observations that you brush in the scatter plot are highlighted in the Variables editor. When a figure is not linked to its data sources, you can still brush its graphs and you can brush the same data in the Variables editor, but only the display that you brush responds by highlighting.

Tips

- As an alternative to the `openvar` function, double-click a variable in the Workspace browser.

See Also

`brush` | `linkdata` | `load` | `workspace` | `save`

Concepts

- “View, Edit, and Copy Variables”
- “Making Graphs Responsive with Data Linking”

Purpose

Optimization options values

Syntax

```
val = optimget(options,'param')
val = optimget(options,'param',default)
```

Description

`val = optimget(options,'param')` returns the value of the specified parameter in the optimization options structure `options`. You need to type only enough leading characters to define the parameter name uniquely. Case is ignored for parameter names.

`val = optimget(options,'param',default)` returns `default` if the specified parameter is not defined in the optimization options structure `options`. Note that this form of the function is used primarily by other optimization functions.

Examples

This statement returns the value of the `Display` optimization options parameter in the structure called `my_options`.

```
val = optimget(my_options,'Display')
```

This statement returns the value of the `Display` optimization options parameter in the structure called `my_options` (as in the previous example) except that if the `Display` parameter is not defined, it returns the value `'final'`.

```
optnew = optimget(my_options,'Display','final');
```

See Also

`optimset` | `fminbnd` | `fminsearch` | `fzero` | `lsqnonneg`

optimset

Purpose

Create or edit optimization options structure

Syntax

```
options = optimset('param1',value1,'param2',value2,...)
optimset
options = optimset
options = optimset(optimfun)
options = optimset(oldopts,'param1',value1,...)
options = optimset(oldopts,newopts)
```

Description

The function `optimset` creates an `options` structure that you can pass as an input argument to the following four MATLAB optimization functions:

- `fminbnd`
- `fminsearch`
- `fzero`
- `lsqnonneg`

You can use the `options` structure to change the default parameters for these functions.

Note If you have an Optimization Toolbox™ license, you can also use `optimset` to create an expanded `options` structure containing additional options specifically designed for the functions provided in that toolbox. For more information about these additional options, see the reference page for the enhanced Optimization Toolbox `optimset` function.

`options = optimset('param1',value1,'param2',value2,...)` creates an optimization options structure called `options`, in which the specified parameters (`param`) have specified values. Any unspecified parameters are set to `[]` (parameters with value `[]` indicate to use the default value for that parameter when `options` is passed to the optimization function). It is sufficient to type only enough leading

characters to define the parameter name uniquely. Case is ignored for parameter names.

`optimset` with no input or output arguments displays a complete list of parameters with their valid values.

`options = optimset` (with no input arguments) creates an options structure `options` where all fields are set to `[]`.

`options = optimset(optimfun)` creates an options structure `options` with all parameter names and default values relevant to the optimization function `optimfun`.

`options = optimset(oldopts,'param1',value1,...)` creates a copy of `oldopts`, modifying the specified parameters with the specified values.

`options = optimset(oldopts,newopts)` combines an existing options structure `oldopts` with a new options structure `newopts`. Any parameters in `newopts` with nonempty values overwrite the corresponding old parameters in `oldopts`.

Options

The following table lists the available options for the MATLAB optimization functions.

Option	Value	Description	Solvers
`Display`	`'off'` | `'iter'` | {`'final'`} | `'notify'`	Level of display. `'off'` displays no output; `'iter'` displays output at each iteration (not available for `lsqnonneg`); `'final'` displays just the final output; `'notify'` displays output only if	`fminbnd`, `fminsearch`,`fzero`, `lsqnonneg`

Option	Value	Description	Solvers
		the function does not converge.	
FunValCheck	{'off'} | 'on'	Check whether objective function values are valid. 'on' displays an error when the objective function returns a value that is complex or NaN. 'off' displays no error.	fminbnd, fminsearch,fzero
MaxFunEvals	positive integer	Maximum number of function evaluations allowed.	fminbnd, fminsearch
MaxIter	positive integer	Maximum number of iterations allowed.	fminbnd, fminsearch
OutputFcn	function | {[]}	User-defined function that an optimization function calls at each iteration.	fminbnd, fminsearch,fzero
PlotFcns	function | {[]}	User-defined or built-in plot function that an optimization function calls at each iteration. Built-in functions: • @optimplotx plots the current point • @optimplotfval plots the function value • @optimplotfunccount plots the function	fminbnd, fminsearch,fzero

Option	Value	Description	Solvers
		count (not available for fzero)	
TolFun	positive scalar	Termination tolerance on the function value.	fminsearch
TolX	positive scalar	Termination tolerance on *x*, the current point.	fminbnd, fminsearch,fzero, lsqnonneg

Examples

This statement creates an optimization options structure called options in which the Display parameter is set to 'iter' and the TolFun parameter is set to 1e-8.

```
options = optimset('Display','iter','TolFun',1e-8)
```

This statement makes a copy of the options structure called options, changing the value of the TolX parameter and storing new values in optnew.

```
optnew = optimset(options,'TolX',1e-4);
```

This statement returns an optimization options structure that contains all the parameter names and default values relevant to the function fminbnd.

```
optimset('fminbnd')
```

See Also

optimset | optimget | fminbnd | fminsearch | fzero | lsqnonneg

Purpose

Find logical OR of array or scalar inputs

Syntax

```
A | B | ...
or(A, B)
```

Description

`A | B | ...` performs a logical OR of all input arrays `A`, `B`, etc., and returns an array containing elements set to either logical 1 (`true`) or logical 0 (`false`). An element of the output array is set to 1 if any input arrays contain a nonzero element at that same array location. Otherwise, that element is set to 0.

Each input of the expression can be an array or can be a scalar value. All nonscalar input arrays must have equal dimensions. If one or more inputs are an array, then the output is an array of the same dimensions. If all inputs are scalar, then the output is scalar.

If the expression contains both scalar and nonscalar inputs, then each scalar input is treated as if it were an array having the same dimensions as the other input arrays. In other words, if input `A` is a 3-by-5 matrix and input `B` is the number 1, then `B` is treated as if it were a 3-by-5 matrix of ones.

`or(A, B)` is called for the syntax `A | B` when either `A` or `B` is an object.

Note The symbols `|` and `||` perform different operations in a MATLAB application. The element-wise OR operator described here is `|`. The short-circuit OR operator is `||`.

Examples

If matrix `A` is

```
0.4235    0.5798         0    0.7942         0
0.5155         0         0         0    0.8744
     0         0         0    0.4451    0.0150
0.4329    0.6405    0.6808         0         0
```

and matrix `B` is

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

then

```
A | B
ans =
     1     1     0     1     0
     1     1     0     0     1
     0     0     0     1     1
     1     1     1     0     1
```

See Also

`bitor` | `and` | `xor` | `not` | `any` | `all`

How To

- logical operators

ordeig

Purpose Eigenvalues of quasitriangular matrices

Syntax

```
E = ordeig(T)
E = ordeig(AA,BB)
```

Description `E = ordeig(T)` takes a quasitriangular Schur matrix `T`, typically produced by `schur`, and returns the vector `E` of eigenvalues in their order of appearance down the diagonal of T.

`E = ordeig(AA,BB)` takes a quasitriangular matrix pair `AA` and `BB`, typically produced by `qz`, and returns the generalized eigenvalues in their order of appearance down the diagonal of `AA-º*BB`.

`ordeig` is an order-preserving version of `eig` for use with `ordschur` and `ordqz`. It is also faster than `eig` for quasitriangular matrices.

Examples

Example 1

```
T=diag([1 -1 3 -5 2]);
```

`ordeig(T)` returns the eigenvalues of `T` in the same order they appear on the diagonal.

```
ordeig(T)

ans =

     1
    -1
     3
    -5
     2
```

`eig(T)`, on the other hand, returns the eigenvalues in order of increasing magnitude.

```
eig(T)

ans =
```

```
    -5
    -1
     1
     2
     3
```

Example 2

```
A = rand(10);
[U, T] = schur(A);
abs(ordeig(T))

ans =

    5.3786
    0.7564
    0.7564
    0.7802
    0.7080
    0.7080
    0.5855
    0.5855
    0.1445
    0.0812
% Move eigenvalues with magnitude < 0.5 to the
% upper-left corner of T.
[U,T] = ordschur(U,T,abs(E)<0.5);
abs(ordeig(T))

ans =

    0.1445
    0.0812
    5.3786
    0.7564
    0.7564
    0.7802
```

```
0.7080
0.7080
0.5855
0.5855
```

See Also

schur | qz | ordschur | ordqz | eig

Purpose

Order fields of structure array

Syntax

```
s = orderfields(s1)
s = orderfields(s1, s2)
s = orderfields(s1, c)
s = orderfields(s1, perm)
[s, perm] = orderfields(...)
```

Description

`s = orderfields(s1)` orders the fields in `s1` so that the new structure array `s` has field names in ASCII dictionary order.

`s = orderfields(s1, s2)` orders the fields in `s1` so that the new structure array `s` has field names in the same order as those in `s2`. Structures `s1` and `s2` must have the same fields.

`s = orderfields(s1, c)` orders the fields in `s1` so that the new structure array `s` has field names in the same order as those in the cell array of field name strings `c`. Structure `s1` and cell array `c` must contain the same field names.

`s = orderfields(s1, perm)` orders the fields in `s1` so that the new structure array `s` has fieldnames in the order specified by the indices in permutation vector `perm`.

If `s1` has `N` fieldnames, the elements of `perm` must be an arrangement of the numbers from `1` to `N`. This is particularly useful if you have more than one structure array that you would like to reorder in the same way.

`[s, perm] = orderfields(...)` returns a permutation vector representing the change in order performed on the fields of the structure array that results in `s`.

Tips

`orderfields` only orders top-level fields. It is not recursive.

Examples

Create a structure `s`. Then create a new structure from `s`, but with the fields ordered alphabetically:

```
s = struct('b', 2, 'c', 3, 'a', 1)
s =
```

```
    b: 2
    c: 3
    a: 1

snew = orderfields(s)
snew =
    a: 1
    b: 2
    c: 3
```

Arrange the fields of `s` in the order specified by the second (cell array) argument of `orderfields`. Return the new structure in `snew` and the permutation vector used to create it in `perm`:

```
[snew, perm] = orderfields(s, {'b', 'a', 'c'})
snew =
    b: 2
    a: 1
    c: 3
perm =
     1
     3
     2
```

Now create a new structure, `s2`, having the same fieldnames as `s`. Reorder the fields using the permutation vector returned in the previous operation:

```
s2 = struct('b', 3, 'c', 7, 'a', 4)
s2 =
    b: 3
    c: 7
    a: 4

snew = orderfields(s2, perm)
snew =
    b: 3
    a: 4
```

```
c: 7
```

See Also

`struct` | `fieldnames` | `setfield` | `getfield` | `isfield` | `rmfield`

How To

- dynamic field names

ordqz

Purpose

Reorder eigenvalues in QZ factorization

Syntax

```
[AAS,BBS,QS,ZS] = ordqz(AA,BB,Q,Z,select)
[...] = ordqz(AA,BB,Q,Z,keyword)
[...] = ordqz(AA,BB,Q,Z,clusters)
```

Description

`[AAS,BBS,QS,ZS] = ordqz(AA,BB,Q,Z,select)` reorders the QZ factorizations `Q*A*Z = AA` and `Q*B*Z = BB` produced by the `qz` function for a matrix pair `(A,B)`. It returns the reordered pair `(AAS,BBS)` and the cumulative orthogonal transformations `QS` and `ZS` such that `QS*A*ZS = AAS` and `QS*B*ZS = BBS`. In this reordering, the selected cluster of eigenvalues appears in the leading (upper left) diagonal blocks of the quasitriangular pair `(AAS,BBS)`, and the corresponding invariant subspace is spanned by the leading columns of `ZS`. The logical vector `select` specifies the selected cluster as `E(select)` where `E` is the vector of eigenvalues as they appear along the diagonal of `AA-º*BB`.

Note To extract `E` from `AA` and `BB`, use `ordeig(BB)`, instead of `eig`. This ensures that the eigenvalues in `E` occur in the same order as they appear on the diagonal of `AA-º*BB`.

`[...] = ordqz(AA,BB,Q,Z,keyword)` sets the selected cluster to include all eigenvalues in the region specified by `keyword`:

keyword	Selected Region
'lhp'	Left-half plane (`real(E) < 0`)
'rhp'	Right-half plane (`real(E) > 0`)
'udi'	Interior of unit disk (`abs(E) < 1`)
'udo'	Exterior of unit disk (`abs(E) > 1`)

`[...] = ordqz(AA,BB,Q,Z,clusters)` reorders multiple clusters at once. Given a vector `clusters` of cluster indices commensurate with `E = ordeig(AA,BB)`, such that all eigenvalues with the same `clusters`

value form one cluster, `ordqz` sorts the specified clusters in descending order along the diagonal of `(AAS,BBS)`. The cluster with highest index appears in the upper left corner.

See Also

`ordeig` | `ordschur` | `qz`

ordschur

Purpose

Reorder eigenvalues in Schur factorization

Syntax

```
[US,TS] = ordschur(U,T,select)
[US,TS] = ordschur(U,T,keyword)
[US,TS] = ordschur(U,T,clusters)
```

Description

`[US,TS] = ordschur(U,T,select)` reorders the Schur factorization `X = U*T*U'` produced by the `schur` function and returns the reordered Schur matrix `TS` and the cumulative orthogonal transformation `US` such that `X = US*TS*US'`. In this reordering, the selected cluster of eigenvalues appears in the leading (upper left) diagonal blocks of the quasitriangular Schur matrix `TS`, and the corresponding invariant subspace is spanned by the leading columns of `US`. The logical vector `select` specifies the selected cluster as `E(select)` where `E` is the vector of eigenvalues as they appear along `T`'s diagonal.

Note To extract `E` from `T`, use `E = ordeig(T)`, instead of `eig`. This ensures that the eigenvalues in `E` occur in the same order as they appear on the diagonal of `TS`.

`[US,TS] = ordschur(U,T,keyword)` sets the selected cluster to include all eigenvalues in one of the following regions:

keyword	Selected Region
`'lhp'`	Left-half plane (`real(E) < 0`)
`'rhp'`	Right-half plane (`real(E) > 0`)
`'udi'`	Interior of unit disk (`abs(E) < 1`)
`'udo'`	Exterior of unit disk (`abs(E) > 1`)

`[US,TS] = ordschur(U,T,clusters)` reorders multiple clusters at once. Given a vector `clusters` of cluster indices, commensurate with `E = ordeig(T)`, and such that all eigenvalues with the same `clusters` value form one cluster, `ordschur` sorts the specified clusters

in descending order along the diagonal of TS, the cluster with highest index appearing in the upper left corner.

See Also

ordeig | ordqz | schur

orient

Purpose

Hardcopy paper orientation

Alternatives

Use **File —> Print Preview** on the figure window menu to directly manipulate print layout, paper size, headers, fonts and other properties when printing figures. For details, see Using Print Preview in the MATLAB Graphics documentation.

Syntax

```
orient
orient landscape
orient portrait
orient tall
orient(fig_handle), orient(simulink_model)
orient(fig_handle,orientation), orient(simulink_model,
   orientation)
```

Description

`orient` returns a string with the current paper orientation: `portrait`, `landscape`, or `tall`.

`orient landscape` sets the paper orientation of the current figure to full-page landscape, orienting the longest page dimension horizontally. The figure is centered on the page and scaled to fit the page with a 0.25 inch border.

`orient portrait` sets the paper orientation of the current figure to portrait, orienting the longest page dimension vertically. The `portrait` option returns the page orientation to the MATLAB default. (Note that the result of using the `portrait` option is affected by changes you make to figure properties. See the "Algorithm" section for more specific information.)

`orient tall` maps the current figure to the entire page in portrait orientation, leaving a 0.25 inch border.

`orient(fig_handle), orient(simulink_model)` returns the current orientation of the specified figure or Simulink model.

`orient(fig_handle,`*`orientation`*`),`
`orient(simulink_model,`*`orientation`*`)` sets the orientation for

the specified figure or Simulink model to the specified orientation (`landscape`, `portrait`, or `tall`).

Algorithms

`orient` sets the `PaperOrientation`, `PaperPosition`, and `PaperUnits` properties of the current figure. Subsequent print operations use these properties. The result of using the `portrait` option can be affected by default property values as follows:

- If the current figure `PaperType` is the same as the default figure `PaperType` and the default figure `PaperOrientation` has been set to `landscape`, then the `orient portrait` command uses the current values of `PaperOrientation` and `PaperPosition` to place the figure on the page.
- If the current figure `PaperType` is the same as the default figure `PaperType` and the default figure `PaperOrientation` has been set to `landscape`, then the `orient portrait` command uses the default figure `PaperPosition` with the x, y and width, height values reversed (i.e., [y,x,height,width]) to position the figure on the page.
- If the current figure `PaperType` is different from the default figure `PaperType`, then the `orient portrait` command uses the current figure `PaperPosition` with the x, y and width, height values reversed (i.e., [y,x,height,width]) to position the figure on the page.

See Also

`print` | `printpreview` | `set` | `PaperOrientation` | `PaperPosition` | `PaperSize` | `PaperType` | `PaperUnits`

orth

Purpose Range space of matrix

Syntax `B = orth(A)`

Description `B = orth(A)` returns an orthonormal basis for the range of `A`. The columns of `B` span the same space as the columns of `A`, and the columns of `B` are orthogonal, so that `B'*B = eye(rank(A))`. The number of columns of `B` is the rank of `A`.

See Also `null` | `svd` | `rank`

Purpose

Optional keyword in switch statement

Syntax

```
switch switch_expression
   case case_expression
      statements
   case case_expression
      statements
    :
   otherwise
      statements
end
```

Description

A `switch` block conditionally executes one set of statements from several choices. Each choice is a `case`.

An evaluated *`switch_expression`* is a scalar or string. An evaluated *`case_expression`* is a scalar, a string, or a cell array of scalars or strings. The `switch` block tests each case until one of the cases is `true`. A case is `true` when:

- For numbers, `eq(`*`case_expression`*`,`*`switch_expression`*`)`.
- For strings, `strcmp(`*`case_expression`*`,`*`switch_expression`*`)`.
- For objects that support the `eq` function, `eq(`*`case_expression`*`,`*`switch_expression`*`)`.
- For a cell array *`case_expression`*, at least one of the elements of the cell array matches *`switch_expression`*, as defined above for numbers, strings, and objects.

When a case is true, MATLAB executes the corresponding statements, and then exits the `switch` block.

`otherwise` is optional, and executes only when no case is true.

For more information, see the `switch` reference page.

Examples

Decide which plot to create based on the value of the string `plottype`:

```
x = [12, 64, 24];
plottype = 'pie3';

switch plottype
    case 'bar'
        bar(x)
        title('Bar Graph')
    case {'pie','pie3'}
        pie3(x)
        title('Pie Chart')
        legend('First','Second','Third')
    otherwise
        warning('Unexpected plot type. No plot created.');
end
```

See Also

`switch` | `end` | `if` | `for` | `while`

Purpose

Consolidate workspace memory

Syntax

```
pack
pack filename
pack('filename')
```

Description

`pack` frees up needed space by reorganizing information so that it only uses the minimum memory required. All variables from your base and global workspaces are preserved. Any persistent variables that are defined at the time are set to their default value (the empty matrix, `[]`).

The MATLAB software temporarily stores your workspace data in a file called `tp######.mat` (where `######` is a numeric value) that is located in your temporary folder. (You can use the command `dir(tempdir)` to see the files in this folder).

`pack filename` frees space in memory, temporarily storing workspace data in a file specified by `filename`. This file resides in your current working folder and, unless specified otherwise, has a `.mat` file extension.

`pack('filename')` is the function form of `pack`.

Tips

You can only run `pack` from the MATLAB command line.

If you specify a `filename` argument, that file must reside in a folder for which you have write permission.

The `pack` function does not affect the amount of memory allocated to the MATLAB process. You must quit MATLAB to free up this memory.

Since MATLAB uses a heap method of memory management, extended MATLAB sessions may cause memory to become fragmented. When memory is fragmented, there may be plenty of free space, but not enough contiguous memory to store a new large variable.

If you get the `Out of memory` message from MATLAB, the `pack` function may find you some free memory without forcing you to delete variables.

The `pack` function frees space by

- Saving all variables in the base and global workspaces to a temporary file.
- Clearing all variables and functions from memory.
- Reloading the base and global workspace variables back from the temporary file and then deleting the file.

If you use `pack` and there is still not enough free memory to proceed, you must clear some variables. If you run out of memory often, you can allocate larger matrices earlier in the MATLAB session and use these system-specific tips:

- When running MATLAB on The Open Group UNIX platforms, ask your system manager to increase your swap space.
- On Microsoft Windowsplatforms, increase virtual memory using the Windows Control Panel.

To maintain persistent variables when you run pack, use `mlock` in the function.

Examples

Change the current folder to one that is writable, run `pack`, and return to the previous folder.

```
cwd = pwd;
cd(tempdir);
pack
cd(cwd)
```

See Also

`clear` | `memory`

Purpose

Padé approximation of time delays

Syntax

```
[num,den] = padecoef(T,N)
```

Description

`[num,den] = padecoef(T,N)` returns the `N`th-order Padé approximation of the continuous-time delay T in transfer function form. The row vectors `num` and `den` contain the numerator and denominator coefficients in descending powers of s. Both are `N`th-order polynomials.

Class support for input T:

`float: double, single`

Class Support

Input T support floating-point values of type `single` or `double`.

References

[1] Golub, G. H. and C. F. Van Loan *Matrix Computations*, 3rd ed. Johns Hopkins University Press, Baltimore: 1996, pp. 572–574.

See Also

`pade`

pagesetupdlg

Purpose

Page setup dialog box

Syntax

```
dlg = pagesetupdlg(fig)
```

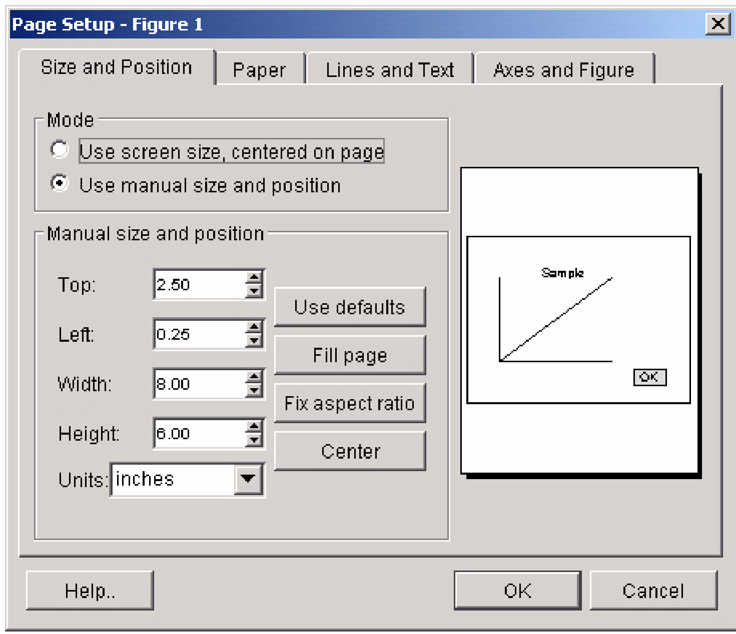
Note `pagesetupdlg` is not recommended. Use `printpreview` instead.

Description

`dlg = pagesetupdlg(fig)` creates a dialog box from which a set of pagelayout properties for the figure window, `fig`, can be set.

`pagesetupdlg` implements the "Page Setup..." option in the **Figure File Menu**.

`pagesetupdlg` supports setting the layout for a single figure. `fig` must be a single figure handle, not a vector of figures or a simulink diagram.



See Also

`printdlg` | `printpreview` | `printopt`

pan

Purpose

Pan view of graph interactively

Syntax

```
pan on
pan xon
pan yon
pan off
pan
pan(figure_handle,...)
h = pan(figure_handle)
```

Description

`pan on` turns on mouse-based panning in the current figure.

`pan xon` turns on panning only in the *x* direction in the current figure.

`pan yon` turns on panning only in the *y* direction in the current figure.

`pan off` turns panning off in the current figure.

`pan` toggles the pan state in the current figure `on` or `off`.

`pan(figure_handle,...)` sets the pan state in the specified figure.

`h = pan(figure_handle)` returns the figure's pan *mode object* for the figure `figure_handle` for you to customize the mode's behavior.

Using Pan Mode Objects

Access the following properties of pan mode objects via `get` and modify some of them using `set`:

- *`Enable`* `'on'|'off'` — Specifies whether this figure mode is currently enabled on the figure
- *`Motion`* `'horizontal'|'vertical'|'both'` — The type of panning enabled for the figure
- `FigureHandle <handle>` — The associated figure handle, a read-only property that cannot be set

Pan Mode Callbacks

You can program the following callbacks for pan mode operations.

- ButtonDownFilter <function_handle> — Function to intercept ButtonDown events

  The application can inhibit the panning operation under circumstances the programmer defines, depending on what the callback returns. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks):

  ```
  function [res] = myfunction(obj,event_obj)
  % obj           handle to the object that has been clicked on
  % event_obj     event data (empty in this release)
  % res [output] a logical flag to determine whether the pan
  %               operation should take place(for 'res' set to 'false'
  %               or the 'ButtonDownFcn' property of the object should
  %               take precedence (when 'res' is 'true')
  ```

- ActionPreCallback <function_handle> — Function to execute before panning

  Set this callback to if you need to execute code when a pan operation begins. The function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks):

  ```
  function myfunction(obj,event_obj)
  % obj          handle to the figure that has been clicked on
  % event_obj    object containing struct of event data
  ```

  The event data struct has the following field:

Axes	The handle of the axes that is being panned

- ActionPostCallback <function_handle> — Function to execute after panning

  Set this callback if you need to execute code when a pan operation ends. The function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks):

```
function myfunction(obj,event_obj)
% obj         handle to the figure that has been clicked on
% event_obj   object containing struct of event data (same as the
%             event data of the 'ActionPreCallback' callback)
```

Pan Mode Utility Functions

The following functions in pan mode query and set certain of its properties.

- `flags = isAllowAxesPan(h,axes)` — Function querying permission to pan axes

  Calling the function `isAllowAxesPan` on the pan object, `h`, with a vector of axes handles, `axes`, as input returns a logical array of the same dimension as the axes handle vector, which indicates whether a pan operation is permitted on the axes objects.

- `setAllowAxesPan(h,axes,flag)` — Function to set permission to pan axes

  Calling the function `setAllowAxesPan` on the pan object, `h`, with a vector of axes handles, `axes`, and a logical scalar, `flag`, either allows or disallows a pan operation on the axes objects.

- *`info`* `= getAxesPanMotion(h,axes)` — Function to get style of pan operations

  Calling the function `getAxesPanMotion` on the pan object, `h`, with a vector of axes handles, `axes`, as input will return a character cell array of the same dimension as the axes handle vector, which indicates the type of pan operation for each axes. Possible values for the type of operation are `'horizontal'`, `'vertical'` or `'both'`.

- `setAxesPanMotion(h,axes,style)` — Function to set style of pan operations

  Calling the function `setAxesPanMotion` on the pan object, `h`, with a vector of axes handles, `axes`, and a character array, `style`, sets the style of panning on each axes.

Examples

Example 1 — Entering Pan Mode

Plot a graph and turn on Pan mode:

```
plot(magic(10));
pan on
% pan on the plot
```

Example 2 — Constrained Pan

Constrain pan to *x*-axis using `set`:

```
plot(magic(10));
h = pan;
set(h,'Motion','horizontal','Enable','on');
% pan on the plot in the horizontal direction.
```

Example 3 — Constrained Pan in Subplots

Create four axes as subplots and give each one a different panning behavior:

```
ax1 = subplot(2,2,1);
plot(1:10);
h = pan;
ax2 = subplot(2,2,2);
plot(rand(3));
setAllowAxesPan(h,ax2,false);
ax3 = subplot(2,2,3);
plot(peaks);
setAxesPanMotion(h,ax3,'horizontal');
ax4 = subplot(2,2,4);
contour(peaks);
setAxesPanMotion(h,ax4,'vertical');
% pan on the plots.
```

Example 4 — Coding a ButtonDown Callback

Create a buttonDown callback for pan mode objects to trigger. Copy the following code to a new file, execute it, and observe panning behavior:

```
function demo
% Allow a line to have its own 'ButtonDownFcn' callback.
hLine = plot(rand(1,10));
set(hLine,'ButtonDownFcn','disp(''This executes'')');
set(hLine,'Tag','DoNotIgnore');
h = pan;
set(h,'ButtonDownFilter',@mycallback);
set(h,'Enable','on');
% mouse click on the line
%
function [flag] = mycallback(obj,event_obj)
% If the tag of the object is 'DoNotIgnore', then return true.
% Indicate what the target is
disp(['Clicked ' get(obj,'Type') ' object'])
objTag = get(obj,'Tag');
if strcmpi(objTag,'DoNotIgnore')
   flag = true;
else
   flag = false;
end
```

Example 5 — Coding Pre- and Post-Callback Behavior

Create callbacks for pre- and post-ButtonDown events for pan mode objects to trigger. Copy the following code to a new file, execute it, and observe panning behavior:

```
function demo
% Listen to pan events
plot(1:10);
h = pan;
set(h,'ActionPreCallback',@myprecallback);
set(h,'ActionPostCallback',@mypostcallback);
set(h,'Enable','on');
```

```
%
function myprecallback(obj,evd)
disp('A pan is about to occur.');
%
function mypostcallback(obj,evd)
newLim = get(evd.Axes,'XLim');
msgbox(sprintf('The new X-Limits are [%.2f %.2f].',newLim));
```

Example 6 — Creating a Context Menu for Pan Mode

Coding a context menu that lets the user to switch to Zoom mode by right-clicking:

```
figure; plot(magic(10));
hCM = uicontextmenu;
hMenu = uimenu('Parent',hCM,'Label','Switch to zoom',...
        'Callback','zoom(gcbf,''on'')');
hPan = pan(gcf);
set(hPan,'UIContextMenu',hCM);
pan('on')
```

You cannot add items to the built-in pan context menu, but you can replace it with your own.

Tips

You can create a pan mode object once and use it to customize the behavior of different axes, as Example 3 illustrates. You can also change its callback functions on the fly.

Note **Do not change figure callbacks within an interactive mode.** While a mode is active (when panning, zooming, etc.), you will receive a warning if you attempt to change any of the figure's callbacks and the operation will not succeed. The one exception to this rule is the figure `WindowButtonMotionFcn` callback, which can be changed from within a mode. Therefore, if you are creating a GUI that updates a figure's callbacks, the GUI should some keep track of which interactive mode is active, if any, before attempting to do this.

When you assign different pan behaviors to different `subplot` axes via a mode object and then link them using the `linkaxes` function, the behavior of the axes you manipulate with the mouse carries over to the linked axes, regardless of the behavior you previously set for the other axes.

Alternatives

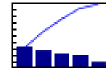
Use the **Pan** tool on the figure toolbar to enable and disable pan mode on a plot, or select **Pan** from the figure's **Tools** menu. For details, see "Panning — Shifting Your View of the Graph".

See Also

`zoom` | `linkaxes` | `rotate3d`

Purpose

Pareto chart



Syntax

```
pareto(Y)
pareto(Y,names)
pareto(Y,X)
H = pareto(...)
```

Description

Pareto charts display the values in the vector `Y` as bars drawn in descending order. Values in `Y` must be nonnegative and not include `NaN`s. Only the first 95% of the cumulative distribution is displayed.

`pareto(Y)` labels each bar with its element index in `Y` and also plots a line displaying the cumulative sum of `Y`.

`pareto(Y,names)` labels each bar with the associated name in the string matrix or cell array `names`.

`pareto(Y,X)` labels each bar with the associated value from `X`.

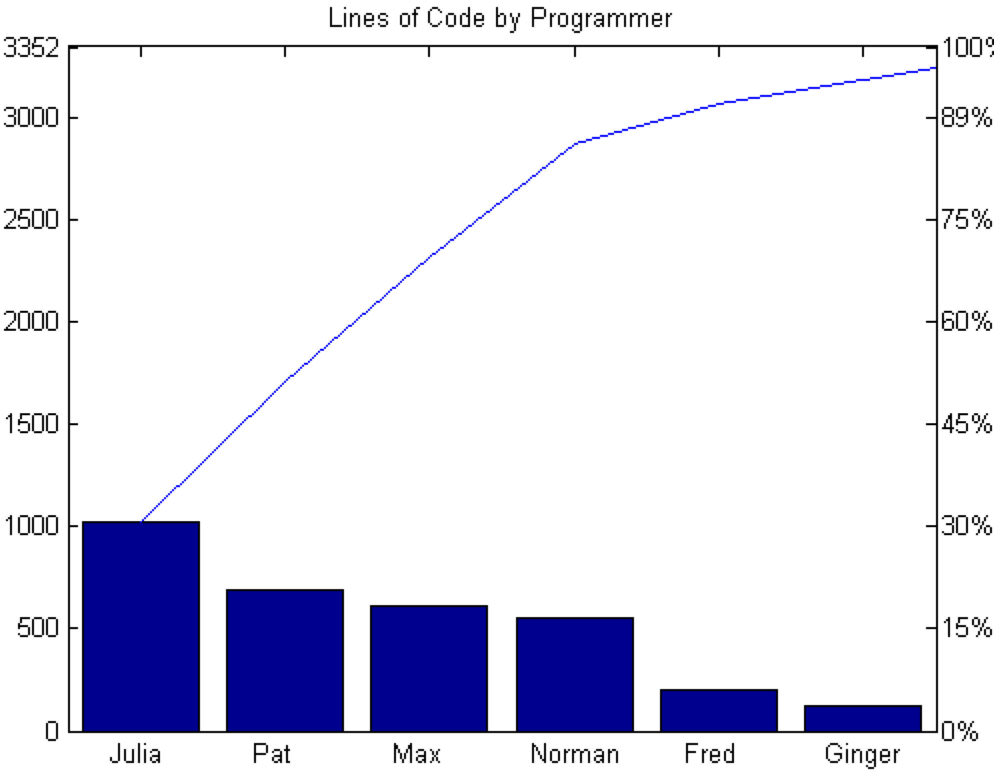
`pareto(ax,..)` plots a Pareto chart in existing axes `ax` rather than `GCA`.

`H = pareto(...)` returns a combination of `patch` and `line` object handles.

Examples

Examine the cumulative productivity of a group of programmers to see how normal its distribution is:

```
codelines = [200 120 555 608 1024 101 57 687];
coders = ...
{'Fred','Ginger','Norman','Max','Julia','Wally','Heidi','Pat'};
pareto(codelines, coders)
title('Lines of Code by Programmer')
```

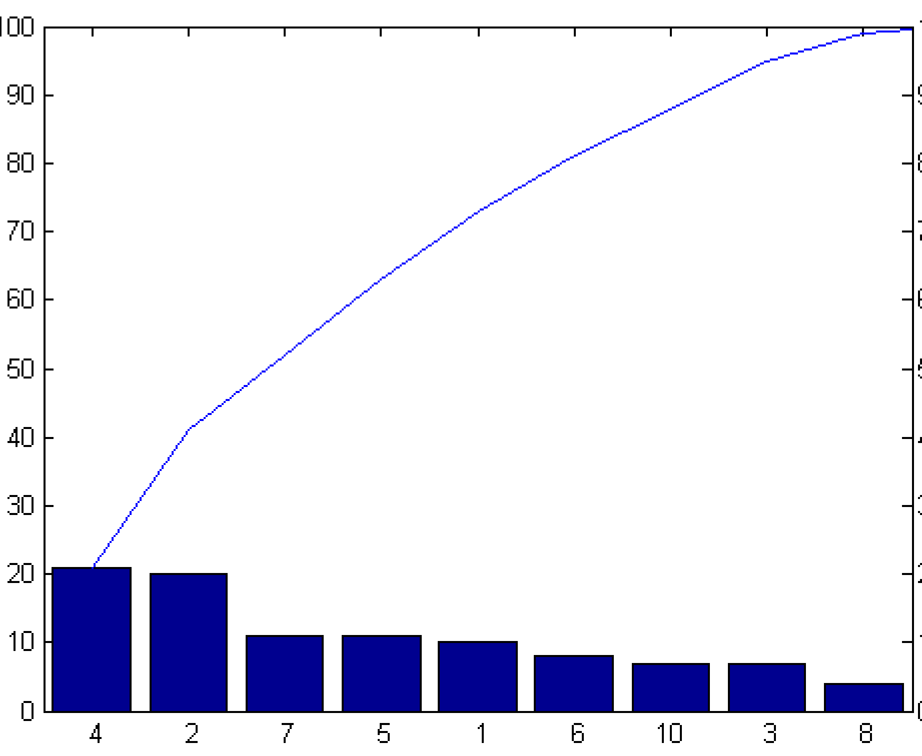


Generate a vector, X, representing diagnostic codes with values from 1 to 10 indicating various faults on devices emerging from a production line:

```
X = min(round(abs(randn(100,1)*4))+1,10);
```

Plot a Pareto chart showing the frequency of failure for each diagnostic code from the most to the least common:

```
pareto(hist(X))
```



Tips

You can use `pareto` to display the output of `hist`, even for vectors that include negative numbers. Because only the first 95 percent of values are displayed, one or more of the smallest bars may not appear. If you extend the `Xlim` of your chart, you can display all the values, but the new bars will not be labeled.

You cannot place datatips (use the Datacursor tool) on graphs created with `pareto`.

See Also

`hist` | `bar`

Purpose

Parallel for loop

Syntax

```
parfor loopvar = initval:endval; statements; end
parfor (loopvar = initval:endval, M); statements; end
```

Description

`parfor loopvar = initval:endval; statements; end` executes a series of MATLAB statements for values of `loopvar` between `initval` and `endval`, inclusive, which specify a vector of increasing integer values. The loop occurs in parallel when you open a pool of workers with Parallel Computing Toolbox™ or when you create a MEX function with MATLAB Coder™. Unlike a traditional `for`-loop, iterations are not executed in a guaranteed order.

`parfor (loopvar = initval:endval, M); statements; end` executes statements in a loop using a maximum of `M` workers or threads, where `M` is a nonnegative integer.

Tips

- If you have Parallel Computing Toolbox software, see the function reference pages for `parfor` and `matlabpool` for additional information.
- If you have MATLAB Coder software, see the `parfor` function reference page for additional information.

Examples

Perform three large eigenvalue computations using three workers or cores with Parallel Computing Toolbox software:

```
matlabpool(3)
parfor i=1:3, c(:,i) = eig(rand(1000)); end
```

See Also

`for`

inputParser.parse

Purpose

Parse function inputs

Syntax

`parse(p,argList)`

Description

`parse(p,argList)` parses and validates the inputs in `arglist`.

Input Arguments

p

Object of class `inputParser`.

argList

Comma separated list of inputs to parse and validate for your custom function. The class of each input depends upon your function definition.

Examples

Input Parsing

Parse and validate required and optional function inputs.

Create a custom function with required and optional inputs in the file `findArea.m`.

```
function a = findArea(width,varargin)
   p = inputParser;
   defaultHeight = 1;
   defaultUnits = 'inches';
   defaultShape = 'rectangle';
   expectedShapes = {'square','rectangle','parallelogram'};

   addRequired(p,'width',@isnumeric);
   addOptional(p,'height',defaultHeight,@isnumeric);
   addParamValue(p,'units',defaultUnits);
   addParamValue(p,'shape',defaultShape,...
                 @(x) any(validatestring(x,expectedShapes)));

   parse(p,width,varargin{:});
   a = p.Results.width .* p.Results.height;
```

The input parser checks whether `width` and `height` are numeric, and whether the `shape` matches a string in cell array `expectedShapes`. `@` indicates a function handle, and the syntax `@(x)` creates an anonymous function with input `x`.

Call the function with inputs that do not match the scheme. For example, specify a nonnumeric value for the `width` input:

```
findArea('text')
```

```
Error using findArea (line 14)
Argument 'width' failed validation isnumeric.
```

Specify an unsupported value for `shape`:

```
findArea(4,'shape','circle')
```

```
Error using findArea (line 14)
Argument 'shape' failed validation with error:
Expected input to match one of these strings:

square, rectangle, parallelogram

The input, 'circle', did not match any of the valid strings.
```

See Also

`addOptional` | `addParamValue` | `addRequired` | `inputParser` |

parseSoapResponse

Purpose

Convert response string from SOAP server into MATLAB types

Syntax

```
parseSoapResponse(response)
```

Description

`parseSoapResponse(response)` extracts data from `response` a string returned by a SOAP server from the `callSoapService` function, and converts it to appropriate MATLAB classes (types).

Examples

This example uses `parseSoapResponse` in conjunction with other SOAP functions to retrieve information about books from a library database, specifically, the author's name for a given book title.

Note The example does not use an actual endpoint; therefore, you cannot run it. The example only illustrates how to use the SOAP functions.

```
% Create the message:
message = createSoapMessage(...
'urn:LibraryCatalog',...
'getAuthor',...
{'In the Fall'},...
{'nameToLookUp'},...
{'{http://www.w3.org/2001/XMLSchema}string'},...
'rpc');
%
% Send the message to the service and get the response:
response = callSoapService(...
'http://test/soap/services/LibraryCatalog',...
'urn:LibraryCatalog#getAuthor',...
message)
%
% Extract MATLAB data from the response
author = parseSoapResponse(response)
```

MATLAB returns:

```
author = Kate Alvin
```

where author is a char class (type).

See Also

`callSoapService` | `createClassFromWsdl` | `createSoapMessage` | `urlread` | `xmlread`

How To

- "Access Web Services Using MATLAB SOAP Functions"

pascal

Purpose

Pascal matrix

Syntax

```
A = pascal(n)
A = pascal(n,1)
A = pascal(n,2)
```

Description

`A = pascal(n)` returns the Pascal matrix of order `n`: a symmetric positive definite matrix with integer entries taken from Pascal's triangle. The inverse of `A` has integer entries.

`A = pascal(n,1)` returns the lower triangular Cholesky factor (up to the signs of the columns) of the Pascal matrix. It is *involutary*, that is, it is its own inverse.

`A = pascal(n,2)` returns a transposed and permuted version of `pascal(n,1)`. `A` is a cube root of the identity matrix.

Examples

`pascal(4)` returns

```
1       1       1       1
1       2       3       4
1       3       6      10
1       4      10      20
```

`A = pascal(3,2)` produces

```
A =
     1     1     1
    -2    -1     0
     1     0     0
```

See Also

`chol`

Purpose

Create one or more filled polygons

Syntax

```
patch(X,Y,C)
patch(X,Y,Z,C)
patch(FV)
patch(X,Y,C,'PropertyName',propertyvalue...)
patch('PropertyName',propertyvalue,...)
handle = patch(...)
```

Properties

For a list of properties, see `Patch Properties`.

Description

`patch(X,Y,C)` adds a filled 2-D patch object to the current axes. A patch object is one or more polygons defined by the coordinates of its vertices. The elements of `X` and `Y` specify the vertices of a polygon . If `X` and `Y` are `m`-by-`n` matrices, MATLAB draws `n` polygons with `m` vertices. `C` determines the color of the patch. For more information on color input requirements, see “Coloring Patches” on page 1-3839.

MATLAB does not require each face to have the same number of vertices. In cases where they do not, pad the end of the `Faces` matrix with `NaNs`. To define a patch with faces that do not close, add one or more `NaNs` to the row in the `Vertices` matrix that defines the vertex you do not want connected.

See “Introduction to Patch Objects” for more information on using patch objects.

`patch(X,Y,Z,C)` creates a patch in 3-D coordinates. If the coordinate data does not define closed polygons, `patch` closes the polygons. The data can define concave or intersecting polygons. However, if the edges of an individual patch face intersect themselves, the resulting face might be only partly filled. In that case, it is better to divide the face into smaller polygons.

`patch(FV)` creates a patch using structure `FV`, which contains the fields `vertices`, `faces`, and optionally `facevertexcdata`. These fields correspond to the `Vertices`, `Faces`, and `FaceVertexCData` patch properties. Specifying only unique vertices and their connection matrix

can reduce the size of the data for patches having many faces. For an example of how to specify patches with this method, see "Specifying Patch Object Shapes" on page 1-3836.

`patch(X,Y,C,'PropertyName',propertyvalue...)` follows the `X`, `Y`, `(Z)`, and `C` arguments with property name/property value pairs to specify additional patch properties. For a description of the properties, see `Patch Properties`. You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the `set` and `get` reference pages for examples of how to specify these data types).

`patch('PropertyName',propertyvalue,...)` specifies all properties using property name/property value pairs. This form lets you omit the color specification because MATLAB uses the default face color and edge color unless you explicitly assign a value to the `FaceColor` and `EdgeColor` properties. This form also lets you specify the patch using the `Faces` and `Vertices` properties instead of *x*-, *y*-, and *z*-coordinates. See "Specifying Patch Object Shapes" on page 1-3836 for more information.

`handle = patch(...)` returns the handle of the patch object it creates.

Unlike high-level area creation functions, such as `fill` or `area`, `patch` does not check the settings of the figure and axes `NextPlot` properties. It simply adds the patch object to the current axes.

Examples

Specifying Patch Object Shapes

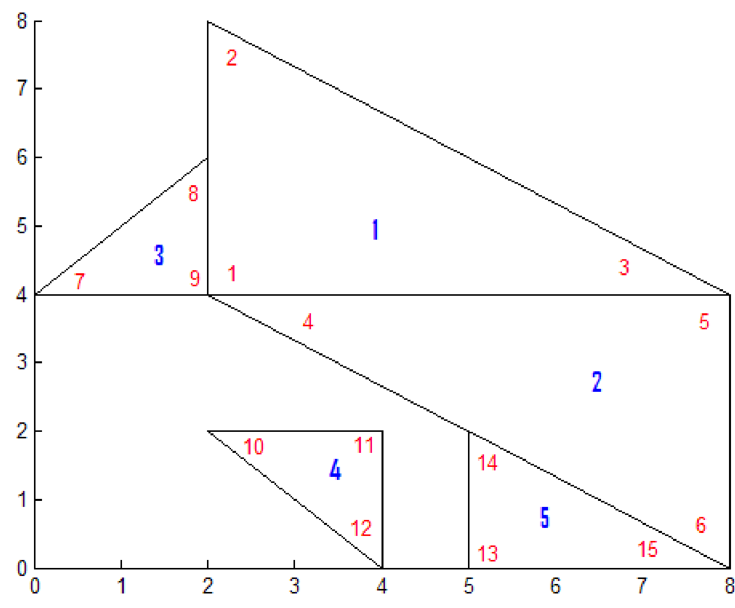
The next two examples create a patch object using two methods:

- Specifying *x*-, *y*-, and *z*-coordinates and color data (`XData`, `YData`, `ZData`, and `CData` properties)
- Specifying vertices, the connection matrix, and color data (`Vertices`, `Faces`, and `FaceVertexCData` properties)

Create five triangular faces, each having three vertices, by specifying the *x*-, *y*-, and *z*-coordinates of each vertex:

```
xdata = [2 2 0 2 5;
         2 8 2 4 5;
```

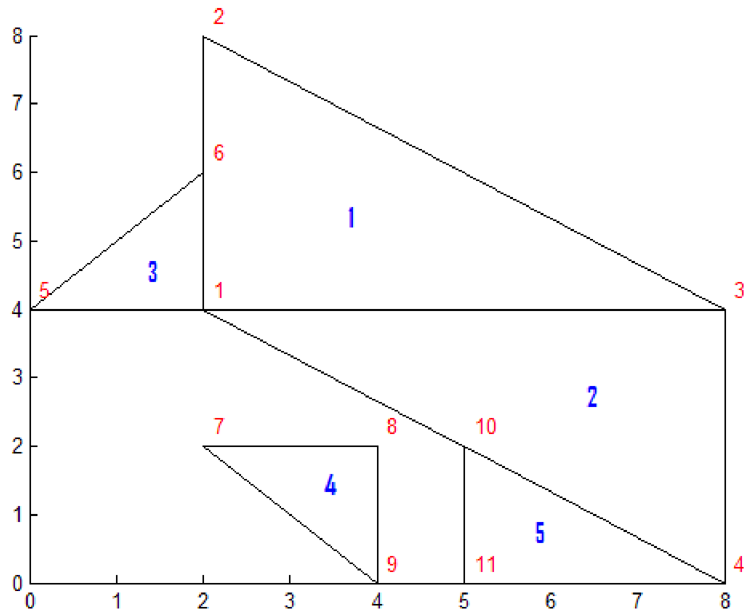
```
         8 8 2 4 8];
ydata = [4 4 4 2 0;
         8 4 6 2 2;
         4 0 4 0 0];
zdata = ones(3,5);

% Red numbers denote the vertex indices.
% For this example:
% xindices = [1 4 7 10 13;
%              2 5 8 11 14;
%              3 6 9 12 15];
% Blue numbers denote the face numbers.
patch(xdata,ydata,zdata,'w')
```



Create the five triangular faces, specifying faces and vertices:

```
% The Vertices property contains the coordinates of each
```

```
% unique vertex defining the patch. The Faces property
% specifies how to connect these vertices to form each
% face of the patch. More than one face may use a given vertex.
% For this example, five triangles have 11 total vertices,
% instead of 15. Each row contains the x- and y-coordinates
% of each vertex.
verts = [2 4; ...
        2 8; ...
        8 4; ...
        8 0; ...
        0 4; ...
        2 6; ...
        2 2; ...
        4 2; ...
        4 0; ...
        5 2; ...
        5 0 ];

% There are five faces, defined by connecting the
% vertices in the order indicated.
faces = [ ...
        1  2  3; ...
        1  3  4; ...
        5  6  1; ...
        7  8  9; ...
        11 10 4 ];

% Create the patch by specifying the Faces, Vertices,
% and FaceVertexCData properties as well as the
% FaceColor property. Red numbers denote the vertex
% numbers, as defined in faces. Blue indicate face numbers.
p = patch('Faces',faces,'Vertices',verts,'FaceColor','w');
```




```
% Using the previous values for verts and faces, you can
% create the same patch object using a structure:
patchinfo.Vertices = verts;
patchinfo.Faces = faces;
patchinfo.FaceColor = 'w';

patch(patchinfo);
```

Coloring Patches

There are many ways to customize your patch objects using colors. The appropriate input depends on:

- Whether you want to change the edge colors
- How you specified the patch faces:
  - Using face/vertex values
  - Using *x*-, *y*-, and *z*-coordinates

The following sections present the various options available.

Specifying Edge Colors

The following options apply to the edge colors of your patch object. The settings are independent of the face colors, but the colors themselves depend on the colors specified at each vertex. Markers show the color at each vertex. Specify the colors using the EdgeColor property. To explore the options using the Sample Input Code, create a base patch object:

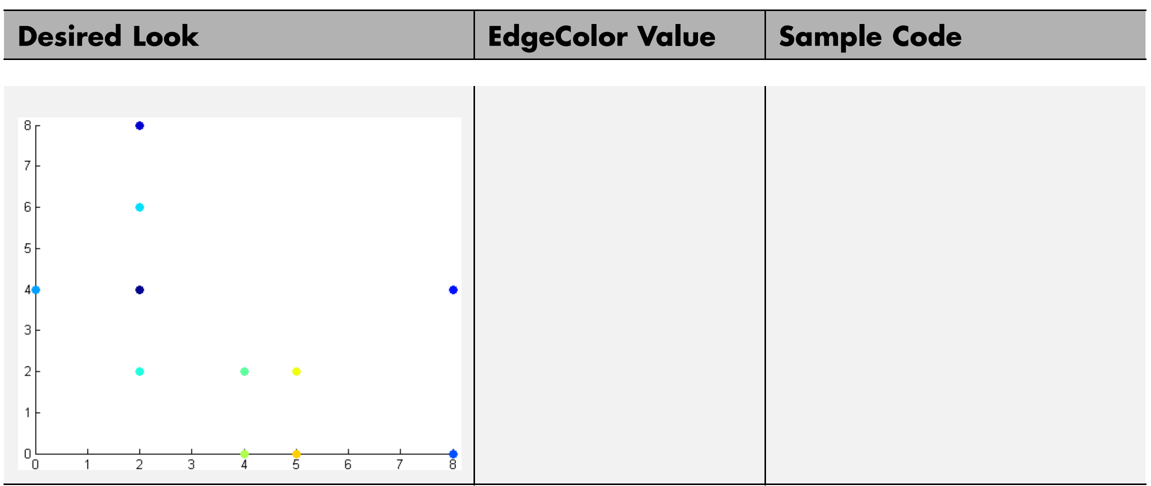
```
xdata = [2     2     0     2     5;
         2     8     2     4     5;
         8     8     2     4     8];
ydata = [4     4     4     2     0;
         8     4     6     2     2;
         4     0     4     0     0];
cdata = [15     0     4     6    10;
         1     2     5     7     9;
         2     3     0     8     3];
p = patch(xdata,ydata,cdata,'Marker','o','MarkerFaceColor','flat','FaceCo
```

For more detailed information on how the EdgeColor property works, see the Patch Properties page.

Desired Look	EdgeColor Value	Sample Code
All edges have the same color, around all faces. This option does not rely on the FaceColor value.	ColorSpec	set(p,'EdgeColor','g')

Desired Look	EdgeColor Value	Sample Code
(continued figure)		
Each edge corresponds to the color of the vertex that precedes the edge, with one color per edge. This option requires that the `FaceColor` property be `flat` or `interp`. By default, if you specify `CData` when creating the patch object, its `FaceColor` property is `interp`.	`'flat'`	`set(p,'EdgeColor','flat',...` `'LineWidth',3)`

Desired Look	EdgeColor Value	Sample Code
Each edge corresponds to the vertex colors, interpolated between vertices. This option requires that the `FaceColor` property be `flat` or `interp`. By default, if you specify `CData` when creating the patch object, its `FaceColor` property is `interp`.	`'interp'`	`set(gcf,'Renderer','zbuffer')` `set(p,'EdgeColor','interp',...` `'LineWidth',5)`
Edges have no color. This option does not rely on the `FaceColor` value. If set, markers retain vertex colors.	`'none'`	`set(p,'EdgeColor','none')`

Desired Look	EdgeColor Value	Sample Code
		

Specifying Face Colors Using Face/Vertex Input Matrics

The following options apply to the face colors of your patch object when you specify the faces using face/vertex input matrices. To explore the options, start with a base patch object:

```
% For this example, there are five triangles (m = 5) sharing
% eleven unique vertices (k = 11).
verts = [2 4; ...
        2 8; ...
        8 4; ...
        8 0; ...
        0 4; ...
        2 6; ...
        2 2; ...
        4 2; ...
        4 0; ...
        5 2; ...
        5 0 ];
faces = [1  2  3; ...
         1  3  4; ...
         5  6  1; ...
         7  8  9; ...
         11 10 4];
```

```
p = patch('Faces',faces,'Vertices',verts,'FaceColor','b');
```

For more information on the relevant properties, see `FaceColor`, `FaceVertexCData`, and `CDataMapping`.

Desired Look	Parameter Values	Sample Code

All faces have the same color.	• `FaceColor`: `ColorSpec`
	• `FaceVertexCData`: `[]` (no input)
	An empty array is the default value, and `patch` ignores any input until you set `FaceColor` to `'flat'` or `'interp'`.
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`set(p,'FaceColor','r')`
		or
		`set(p,'FaceColor',[1 0 0])`

Each face has a single, unique color, indexed from a selected section of the colormap.	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: m-by-1 matrix of index values
	• Color source: A selected portion of the `colormap`
	• `CDataMapping`: `'scaled'`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60]';`
		`set(p,'FaceColor','flat',...`
		`'FaceVertexCData',cdata,...`
		`'CDataMapping','scaled')`

Desired Look	Parameter Values	Sample Code

Each face has a single, unique color, indexed from the whole colormap.	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: m-by-1 matrix of index values
	• Color source: `colormap`
	• `CDataMapping`: `'direct'`
	`'scaled'` is the default value when you input `CData` values. If you want to change the axes `CLim` property, but want your patch object to index the entire colormap, use `'CDataMapping','direct'`.
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60]';`
		`set(p,'FaceColor','flat',...`
		`'FaceVertexCData',cdata,...`
		`'CDataMapping','direct')`

Each face has a single, unique color, determined by truecolor value input.	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: m-by-3 matrix of truecolor values, from 0 to 1
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`clear cdata`
		`cdata = [0 0 1 0 0.8;`
		`0 1 0 0 0.8;`
		`1 0 1 0 0.8]';`
		`set(p,'FaceColor','flat',...`
		`'FaceVertexCData',cdata)`

<table>
<tr><th>Desired Look</th><th>Parameter Values</th><th>Sample Code</th></tr>
<tr><td>Each unique vertex has a single, unique color, indexed from a selected section of the colormap. Faces each have a single, unique color, but edges may have 'flat' or 'interp' color.</td><td>

- FaceColor: 'flat'
- FaceVertexCData: k-by-1 matrix of index values
- Color source: A selected portion of the colormap
- CDataMapping: 'scaled'

</td><td>

```
clear cdata
set(gca,'CLim',[0 40])
cdata = [15 30 25 2 ...
60 12 23 40 13 26 24]';
set(p,'FaceColor','flat',...
'FaceVertexCData',cdata,...
'EdgeColor','flat',...
'LineWidth',5,...
'CDataMapping','scaled')
```

</td></tr>
<tr><td>Each unique vertex has a single, unique color, indexed from the whole colormap. Faces each have a single, unique color, but edges may have 'flat' or 'interp' color.</td><td>

- FaceColor: 'flat'
- FaceVertexCData: k-by-1 matrix of index values
- Color source: colormap
- CDataMapping: 'direct'

  'scaled' is the default value when you input CData values. If you want to change the axes CLim property, but want your patch object to index the entire colormap, use 'CDataMapping','direct'.

</td><td>

```
clear cdata
set(gca,'CLim',[0 40])
cdata = [15 30 25 2 ...
60 12 23 40 13 26 24]';
set(p,'FaceColor','flat',...
'FaceVertexCData',cdata,...
'CDataMapping','direct',...
'EdgeColor','flat',...
'LineWidth',5)
```

</td></tr>
</table>

Desired Look	Parameter Values	Sample Code

Each unique vertex has a single, unique color, determined by truecolor value input. Faces each have a single, unique color, but edges may have `'flat'` or `'interp'` color.	• `FaceColor`: `'flat'`
	• `FaceVertexCData`: k-by-3 matrix of truecolor values, from 0 to 1
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`clear cdata`
		`cdata = [0 0 1;`
		`0 1 0;`
		`0 1 1;`
		`1 0 0;`
		`1 0 1;`
		`1 1 0;`
		`0 0 0;`
		`0.2 0.2 0.2;`
		`0.4 0.4 0.4;`
		`0.6 0.6 0.6;`
		`0.8 0.8 0.8];`
		`set(p,'FaceColor','flat',...`
		`'FaceVertexCData',cdata,...`
		`'EdgeColor','interp',...`
		`'LineWidth',5)`

Each unique vertex has a single, unique color, indexed from a selected section of the colormap. Edges may have `'flat'` or `'interp'` color.	• `FaceColor`: `'interp'`
	• `FaceVertexCData`: k-by-1 matrix of index values
	• Color source: A selected portion of the `colormap`
	• `CDataMapping`: `'scaled'`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 ...`
		`60 12 23 40 13 26 24]';`
		`set(p,'FaceColor','interp',...`
		`'FaceVertexCData',cdata,...`
		`'EdgeColor','flat',...`
		`'LineWidth',5,...`
		`'CDataMapping','scaled')`

<table>
<tr><th>Desired Look</th><th>Parameter Values</th><th>Sample Code</th></tr>
<tr><td>Each unique vertex has a single, unique color, indexed from the whole colormap. Edges may have 'flat' or 'interp' color.</td><td>

FaceColor: 'interp'
FaceVertexCData: k-by-1 matrix of index values
Color source: colormap
CDataMapping: 'direct'

'scaled' is the default value when you input CData values. If you want to change the axes CLim property, but want your patch object to index the entire colormap, use 'CDataMapping','direct'.
</td><td><pre>clear cdata
set(gca,'CLim',[0 40])
cdata = [15 30 25 2 ...
 60 12 23 40 13 26 24]';
set(p,'FaceColor','interp',...
'FaceVertexCData',cdata,...
'CDataMapping','direct',...
'EdgeColor','flat',...
'LineWidth',5)</pre></td></tr>
<tr><td>Each unique vertex has a single, unique color, determined by truecolor value input. Edges may have 'flat' or 'interp' color.</td><td>

FaceColor: 'interp'
FaceVertexCData: k-by-3 matrix of truecolor values, from 0 to 1
Color source: truecolor
CDataMapping: 'direct' or 'scaled'.

'scaled' is the default value, but neither affects the outcome.
</td><td><pre>clear cdata
cdata = [0 0 1;
         0 1 0;
         0 1 1;
         1 0 0;
         1 0 1;
         1 1 0;
         0 0 0;
         0.2 0.2 0.2;
         0.4 0.4 0.4;
         0.6 0.6 0.6;
         0.8 0.8 0.8];
set(p,'FaceColor','interp',...
'FaceVertexCData',cdata,...
'EdgeColor','interp',...
'LineWidth',5)</pre></td></tr>
</table>

Specifying Face Colors Using x-, y-, and z-Coordinate Input

The following options apply to the face colors of your patch object when you specify the faces using *x*-, *y*-, and *z*-coordinates. To explore the options, start with a base patch object:

```
% For this example, there are five (m=5) triangles (n=3).
% The total number of vertices is mxn, or k = 15.
xdata = [2 2 0 2 5;
         2 8 2 4 5;
         8 8 2 4 8];
ydata = [4 4 4 2 0;
         8 4 6 2 2;
         4 0 4 0 0];
zdata = ones(3,5);
p = patch(xdata,ydata,zdata,'b')
```

For more information on the relevant properties, see `FaceColor`, `CData`, and `CDataMapping`.

Desired Look	Parameter Values	Sample Code

All faces have the same color.	• `FaceColor`: `ColorSpec`
	• `CData`: `[]` (no input)
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`set(p,'FaceColor','r')`
		or
		`set(p,'FaceColor',[1 0 0])`

Each face has a single, unique color, indexed from a selected section of the colormap.	• `FaceColor`: `'flat'`
	• `CData`: m-by-1 matrix of index values
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60];`

Desired Look	Parameter Values	Sample Code

	• Color source: A selected portion of the `colormap`
	• `CDataMapping`: `'scaled'`
		`set(p,'FaceColor','flat',...`
		`'CData',cdata,...`
		`'CDataMapping','scaled')`

Each face has a single, unique color, indexed from the whole colormap.	• `FaceColor`: `'flat'`
	• `CData`: `m`-by-1 matrix of index values
	• Color source: `colormap`
	• `CDataMapping`: `'direct'`
	`'scaled'` is the default value when you input `CData` values. If you want to change the axes `CLim` property, but want your patch object to index the entire colormap, use `'CDataMapping','direct'`.
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60];`
		`set(p,'FaceColor','flat',...`
		`'CData',cdata,...`
		`'CDataMapping','direct')`

Desired Look	Parameter Values	Sample Code

Each face has a single, unique color, determined by truecolor value input.	• `FaceColor`: `'flat'`
	• `CData`: m-by-1-by-3 matrix of truecolor values, from 0 to 1
	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`clear cdata`
		`cdata(:,:,1) = [0 0 1 0 0.8];`
		`cdata(:,:,2) = [0 0 0 0 0.8];`
		`cdata(:,:,3) = [1 1 1 0 0.8];`
		`set(p,'FaceColor','flat',...`
		`'CData',cdata)`

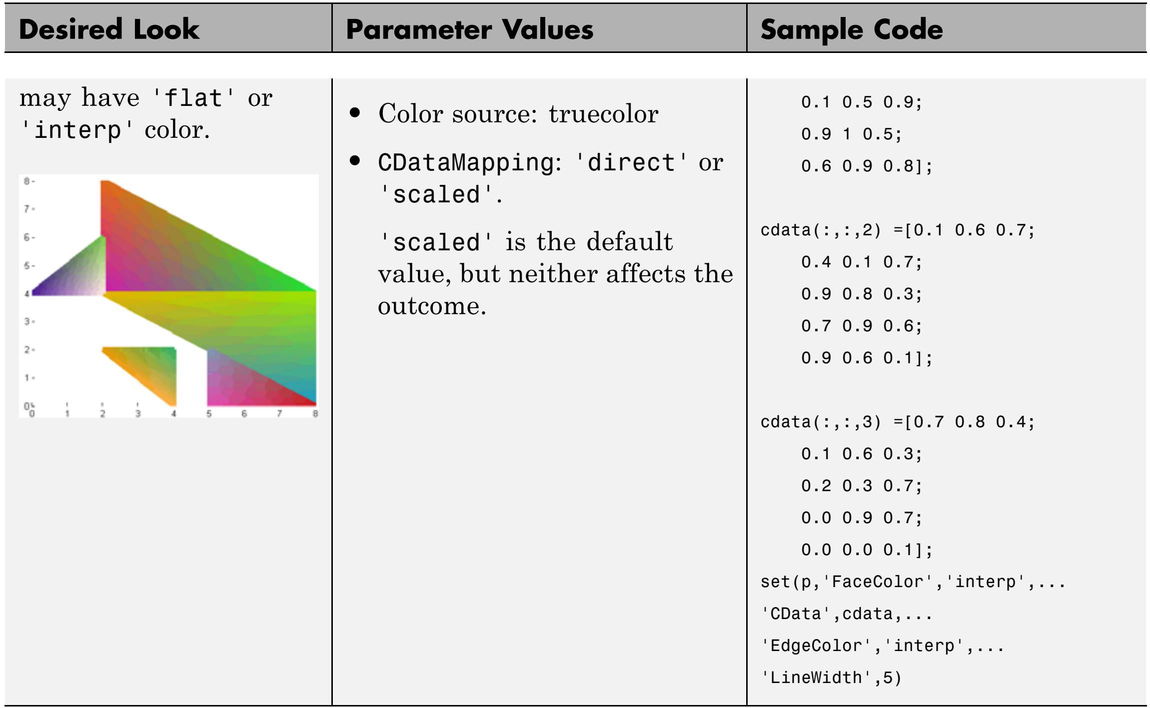
Each unique vertex has a single, unique color, indexed from a selected section of the colormap. Faces each have a single, unique color, but edges may have `'flat'` or `'interp'` color.	• `FaceColor`: `'flat'`
	• `CData`: m-by-n matrix of index values
	• Color source: A selected portion of the `colormap`
	• `CDataMapping`: `'scaled'`
		`clear cdata`
		`set(gca,'CLim',[0 40])`
		`cdata = [15 30 25 2 60;`
		`12 23 40 13 26;`
		`24 8 1 65 42];`
		`set(p,'FaceColor','flat',...`
		`'CData',cdata,...`
		`'EdgeColor','flat',...`
		`'LineWidth',5,...`
		`'CDataMapping','scaled')`

<table>
<tr><th>Desired Look</th><th>Parameter Values</th><th>Sample Code</th></tr>
<tr><td>Each unique vertex has a single, unique color, indexed from the whole colormap. Faces each have a single, unique color, but edges may have <code>'flat'</code> or <code>'interp'</code> color.</td><td><code>FaceColor</code>: <code>'flat'</code>
<code>CData</code>: <code>m</code>-by-<code>n</code> matrix of index values
Color source: <code>colormap</code>
<code>CDataMapping</code>: <code>'direct'</code>
<code>'scaled'</code> is the default value when you input <code>CData</code> values. If you want to change the axes <code>CLim</code> property, but want your patch object to index the entire colormap, use <code>'CDataMapping','direct'</code>.</td><td><pre>clear cdata
set(gca,'CLim',[0 40])
cdata = [15 30 25 2 60;
         12 23 40 13 26;
         24 8 1 65 42];
set(p,'FaceColor','flat',...
'CData',cdata,...
'CDataMapping','direct',...
'EdgeColor','flat',...
'LineWidth',5)</pre></td></tr>
<tr><td>Each vertex has a single, unique color, determined by truecolor value input. Faces each have a single, unique color, but edges may have <code>'flat'</code> or <code>'interp'</code> color.</td><td><code>FaceColor</code>: <code>'flat'</code>
<code>CData</code>: <code>m</code>-by-<code>n</code>-by-3 matrix of truecolor values, from 0 to 1
Color source: truecolor
<code>CDataMapping</code>: <code>'direct'</code> or <code>'scaled'</code>.
<code>'scaled'</code> is the default value, but neither affects the outcome.</td><td><pre>clear cdata
cdata(:,:,1) = [0 0 1 0 0.8;
                0 0 1 0.2 0.6;
                0 1 0 0.4 1];

cdata(:,:,2) = [0 0 0 0 0.8;
                1 1 1 0.2 0.6;
                1 0 0 0.4 0];

cdata(:,:,3) = [1 1 1 0 0.8;
                0 1 0 0.2 0.6;
                1 0 1 0.4 0];
set(p,'FaceColor','flat',...
'CData',cdata,...
'EdgeColor','interp',...
'LineWidth',5)</pre></td></tr>
</table>

Desired Look	Parameter Values	Sample Code
Each vertex has a single, unique color, indexed from a selected section of the colormap. Edges may have 'flat' or 'interp' color.	• FaceColor: 'interp' • CData: m-by-n matrix of index values • Color source: A selected portion of the colormap • CDataMapping: 'scaled'	clear cdata set(gca,'CLim',[0 40]) cdata = [15 30 25 2 60; 12 23 40 13 26; 24 8 1 65 42]; set(p,'FaceColor','interp',... 'CData',cdata,... 'EdgeColor','flat',... 'LineWidth',5,... 'CDataMapping','scaled')
Each vertex has a single, unique color, indexed from the whole colormap. Edges may have 'flat' or 'interp' color.	• FaceColor: 'interp' • CData: m-by-n matrix of index values • Color source: colormap • CDataMapping: 'direct' 'scaled' is the default value when you input CData values. If you want to change the axes CLim property, but want your patch object to index the entire colormap, use 'CDataMapping','direct'.	clear cdata set(gca,'CLim',[0 40]) cdata = [15 30 25 2 60; 12 23 40 13 26; 24 8 1 65 42]; set(p,'FaceColor','interp',... 'CData',cdata,... 'CDataMapping','direct',... 'EdgeColor','flat',... 'LineWidth',5)
Each vertex has a single, unique color, determined by truecolor value input. Edges	• FaceColor: 'interp' • CData: m-by-n-by-3 matrix of truecolor values, from 0 to 1	clear cdata cdata(:,:,1) = [0.8 0.1 0.2 0.9 0.3 1;



Desired Look	Parameter Values	Sample Code

may have `'flat'` or `'interp'` color.	• Color source: truecolor
	• `CDataMapping`: `'direct'` or `'scaled'`.
	`'scaled'` is the default value, but neither affects the outcome.
		`0.1 0.5 0.9;`
		`0.9 1 0.5;`
		`0.6 0.9 0.8];`
		
		`cdata(:,:,2) =[0.1 0.6 0.7;`
		`0.4 0.1 0.7;`
		`0.9 0.8 0.3;`
		`0.7 0.9 0.6;`
		`0.9 0.6 0.1];`
		
		`cdata(:,:,3) =[0.7 0.8 0.4;`
		`0.1 0.6 0.3;`
		`0.2 0.3 0.7;`
		`0.0 0.9 0.7;`
		`0.0 0.0 0.1];`
		`set(p,'FaceColor','interp',...`
		`'CData',cdata,...`
		`'EdgeColor','interp',...`
		`'LineWidth',5)`

See Also

`area` | `caxis` | `fill` | `fill3` | `isosurface` | `surface` | `FaceColor` | `CData` | `CDataMapping` | `FaceVertexCData` | `Patch Properties`

Tutorials

- "Introduction to Patch Objects"

Purpose

Patch properties

Creating Patch Objects

Use `patch` to create patch objects.

Modifying Properties

You can set and query graphics object properties in two ways:

- "The Property Editor" is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see "Setting Default Property Values".

See "Core Graphics Objects" for general information about this type of object.

Patch Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`AlphaDataMapping`
`none| {scaled} | direct`

Transparency mapping method. Determines how the MATLAB software interprets indexed alpha data.

- `none` — The transparency values of `FaceVertexAlphaData` are between 0 and 1 or are clamped to this range.
- `scaled` — Transform the `FaceVertexAlphaData` to span the portion of the alphamap indicated by the axes `ALim` property, linearly mapping data values to alpha values.
- `direct` — Use the `FaceVertexAlphaData` as indices directly into the alphamap. When not scaled, the data are usually integer values ranging from `1` to `length(alphamap)`. MATLAB maps values less than `1` to the first alpha value in the

alphamap, and values greater than `length(alphamap)` to the last alpha value in the alphamap. Values with a decimal portion are fixed to the nearest lower integer. If `FaceVertexAlphaData` is an array of `uint8` integers, then the indexing begins at `0` (that is, MATLAB maps a value of `0` to the first alpha value in the alphamap).

`AmbientStrength`

scalar >= 0 and <= 1

Strength of ambient light. Sets the strength of the ambient light, which is a nondirectional light source that illuminates the entire scene. You must have at least one visible light object in the axes for the ambient light to be visible. The axes `AmbientLightColor` property sets the color of the ambient light, which is therefore the same on all objects in the axes.

You can also set the strength of the diffuse and specular contribution of `light` objects. See the patch `DiffuseStrength` and `SpecularStrength` properties.

`Annotation`

`hg.Annotation` object (read-only)

Handle of `Annotation` object. The `Annotation` property enables you to specify whether this patch object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the patch object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
on	Represent this patch object in a legend (default)
off	Do not include this patch object in a legend
children	Same as on because patch objects do not have children

Setting the IconDisplayStyle property

Set the IconDisplayStyle of a graphics object with handle hobj to off:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Using the IconDisplayStyle property

See "Controlling Legends" for more information and examples.

Selecting which objects to display in legend

Some graphics functions create multiple objects. For example, contour3 uses patch objects to create a 3D contour graph. You can use the Annotation property set select a subset of the objects for display in the legend.

```
[X,Y] = meshgrid(-2:.1:2);
[Cm hC] = contour3(X.*exp(-X.^2-Y.^2));
hA = get(hC,'Annotation');
hLL = get([hA{:}],'LegendInformation');
% Set the IconDisplayStyle property to display
% the first, fifth, and ninth patch in the legend
set([hLL{:}],{'IconDisplayStyle'},...
   {'on','off','off','off','on','off','off','off','on'}')
```

```
% Assign DisplayNames for the three patch
that are displayed in the legend
set(hC([1,5,9]),{'DisplayName'},{'bottom','middle','top'}')
legend show
```

`BackFaceLighting`

`unlit | lit | {reverselit}`

Face lighting control. Determines how faces are lit when their vertex normals point away from the camera.

- `unlit` — Face not lit.
- `lit` — Face lit in normal way.
- `reverselit` — Face lit as if the vertex pointed towards the camera.

Use this property to discriminate between the internal and external surfaces of an object. See "Back Face Lighting" for an example.

`BeingDeleted`

`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Button press callback function. Executes whenever you press a mouse button while the pointer is over the patch object.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

Set this property to a function handle that references the callback. You can also use a string that is a valid MATLAB expression or the name of a MATLAB file. The expressions execute in the MATLAB workspace.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

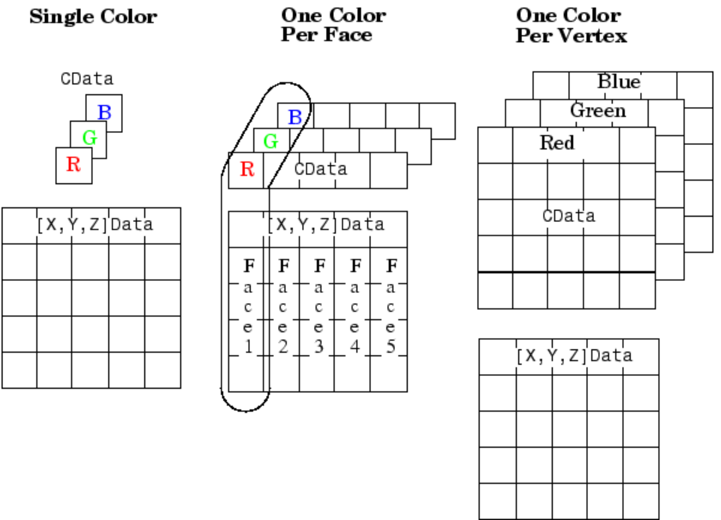
`CData`

scalar | vector | matrix

Patch colors. Specifies the color of the patch. You can specify color for each vertex, each face, or a single color for the entire patch. The way MATLAB interprets `CData` depends on the type of data supplied. The data can be numeric values that are scaled to map linearly into the current colormap, integer values that are used directly as indices into the current colormap, or arrays of RGB values. RGB values are not mapped into the current colormap, but interpreted as the colors defined. On true color systems, MATLAB uses the actual colors defined by the RGB triples.

The following diagrams illustrate the dimensions of `CData` with respect to the coordinate data arrays, `XData`, `YData`, and `ZData`. The first diagram illustrates the use of indexed color.

The following diagram illustrates the use of true color. True color requires `m`-by-`n`-by-`3` arrays to define red, green, and blue components for each color.



Note that if `CData` contains NaNs, MATLAB does not color the faces.

See the `Faces`, `Vertices`, and `FaceVertexCData` properties for an alternative method of patch definition.

`CDataMapping`

`{scaled} | direct`

Direct or scaled color mapping. Determines how MATLAB interprets indexed color data used to color the patch. If you use true color specification for `CData` or `FaceVertexCData`, this property has no effect.

- `scaled` — Transform the color data to span the portion of the colormap indicated by the axes `CLim` property, linearly mapping data values to colors. See the `caxis` command for more information on this mapping.
- `direct` — Use the color data as indices directly into the colormap. When not scaled, the data are usually integer values

ranging from `1` to `length(colormap)`. MATLAB maps values less than `1` to the first color in the colormap, and values greater than `length(colormap)` to the last color in the colormap. Values with a decimal portion are fixed to the nearest lower integer.

`Children`
matrix of handles

Always the empty matrix; patch objects have no children.

`Clipping`
`{on} | off`

Clipping to axes rectangle. When `Clipping` is `on`, MATLAB does not display any portion of the patch outside the axes rectangle.

`CreateFcn`
string | function handle

Callback routine executed during object creation. Executes when MATLAB creates a patch object. You must define this property as a default value for patches or in a call to the `patch` function that creates a new object.

For example, the following statement creates a patch (assuming `x`, `y`, `z`, and `c` are defined), and executes the function referenced by the function handle `@myCreateFcn`.

```
patch(x,y,z,c,'CreateFcn',@myCreateFcn)
```

MATLAB executes the create function after setting all properties for the patch created. Setting this property on an existing patch object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DeleteFcn`
string | function handle

Delete patch callback routine. Executes when you delete the patch object (for example, when you issue a `delete` command or clear the axes (`cla`) or figure (`clf`) containing the patch). MATLAB executes the routine before deleting the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DiffuseStrength`
scalar >= 0 and <= 1

Intensity of diffuse light. Sets the intensity of the diffuse component of the light falling on the patch. Diffuse light comes from light objects in the axes.

You can also set the intensity of the ambient and specular components of the light on the object. See the `AmbientStrength` and `SpecularStrength` properties.

`DisplayName`
string

String used by legend. The `legend` function uses the `DisplayName` property to label the patch object in the legend. The default is an empty string.

- If you specify string arguments with the `legend` function, MATLAB set `DisplayName` to the corresponding string and uses that string for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, MATLAB set `DisplayName` to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add a legend programmatically that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more information and examples.

`EdgeAlpha`

`{scalar = 1} | flat | interp`

Transparency of the edges of patch faces.

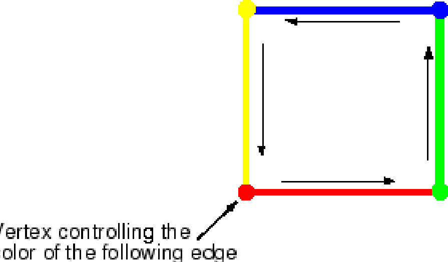
- scalar — A single non-NaN scalar value between `0` and `1` that controls the transparency of all the edges of the object. `1` (the default) means fully opaque and `0` means completely transparent.
- `flat` — The alpha data (`FaceVertexAlphaData`) of each vertex controls the transparency of the edge that follows it.
- `interp` — Linear interpolation of the alpha data (`FaceVertexAlphaData`) at each vertex determines the transparency of the edge.

Note that you cannot specify `flat` or `interp` `EdgeAlpha` without first setting `FaceVertexAlphaData` to a matrix containing one alpha value per face (`flat`) or one alpha value per vertex (`interp`).

`EdgeColor`
`{ColorSpec} | none | flat | interp`

Color of the patch edge. Determines how MATLAB colors the edges of the individual faces that make up the patch.

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for edges. The default value is `[0 0 0]` (`black`). See the `ColorSpec` reference page for more information on specifying color.
- `none` — Edges not drawn.
- `flat` — The color of each vertex controls the color of the edge that follows it. This means `flat` edge coloring is dependent on the order in which you specify the vertices:



- `interp` — Linear interpolation of the `CData` or `FaceVertexCData` values at the vertices determines the edge color.

`EdgeLighting`
`{none} | flat | gouraud | phong`

Algorithm used for lighting calculations. Selects the algorithm used to calculate the effect of light objects on patch edges.

- `none` — Lights do not affect the edges of this object.
- `flat` — The effect of light objects is uniform across each edge of the patch.

- gouraud — The effect of light objects is calculated at the vertices and then linearly interpolated across the edge lines.
- phong — The effect of light objects is determined by interpolating the vertex normals across each edge line and calculating the reflectance at each pixel. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

EraseMode
{normal} | none | xor | background

Erase mode. Controls the technique MATLAB uses to draw and erase patch objects. Alternative erase modes are useful in creating animated sequences, where control of the way individual objects redraw is necessary to improve performance and obtain the desired effect.

- normal — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- none — Do not erase the patch when it is moved or destroyed. While the object is still visible on the screen after erasing with EraseMode none, you cannot print it because MATLAB stores no information about its former location.
- xor — Draw and erase the patch by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the patch does not damage the color of the objects behind it. However, patch color depends on the color of the screen behind it and is correctly colored only when over the axes background Color, or the figure background Color if the axes Color is none.
- background — Erase the patch by drawing it in the axes background Color, or the figure background Color if the axes

`Color` is `none`. This damages objects that are behind the erased patch, but the patch is always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors (for example, performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`FaceAlpha`

{scalar = `1`} `| flat | interp`

Transparency of the patch face.

- scalar — A single non-NaN value between `0` and `1` that controls the transparency of all the faces of the object. `1` (the default) means fully opaque and `0` means completely transparent (invisible).
- `flat` — The values of the alpha data (`FaceVertexAlphaData`) determine the transparency for each face. The alpha data at the first vertex determines the transparency of the entire face.
- `interp` — Bilinear interpolation of the alpha data (`FaceVertexAlphaData`) at each vertex determines the transparency of each face.

Note that you cannot specify `flat` or `interp` `FaceAlpha` without first setting `FaceVertexAlphaData` to a matrix containing one alpha value per face (`flat`) or one alpha value per vertex (`interp`).

`FaceColor`
`{ColorSpec} | none | flat | interp`

Color of the patch face.

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for faces. See the `ColorSpec` reference page for more information on specifying color.
- `none` — Do not draw faces. Note that edges are drawn independently of faces.
- `flat` — The `CData` or `FaceVertexCData` property must contain one value per face and determines the color for each face in the patch. The color data at the first vertex determines the color of the entire face.
- `interp` — Bilinear interpolation of the color at each vertex determines the coloring of each face. The `CData` or `FaceVertexCData` property must contain one value per vertex.

`FaceLighting`
`{none} | flat | gouraud | phong`

Algorithm used for lighting calculations. Selects the algorithm used to calculate the effect of `light` objects on patch faces.

- `none` — Lights do not affect the faces of this object.
- `flat` — The effect of light objects is uniform across the faces of the patch. Select this choice to view faceted objects.
- `gouraud` — The effect of light objects is calculated at the vertices and then linearly interpolated across the faces. Select this choice to view curved surfaces.
- `phong` — The effect of light objects is determined by interpolating the vertex normals across each face and calculating the reflectance at each pixel. Select this choice to

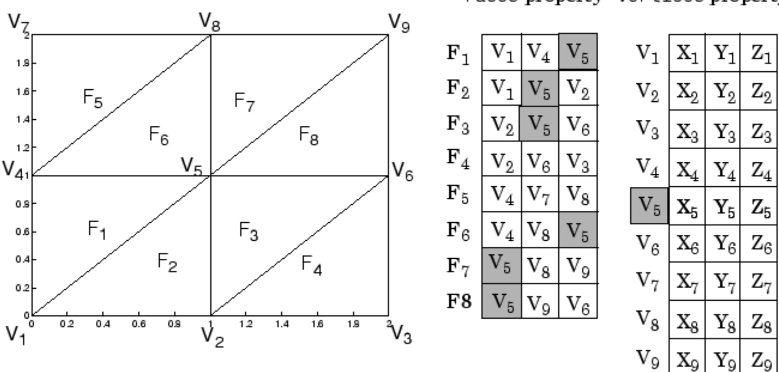
view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

`Faces`

`m`-by-`n` matrix

Vertex connection defining each face. Specifies which vertices in the `Vertices` property are connected. The `Faces` matrix defines *m* faces with up to *n* vertices each. Each row designates the connections for a single face, and the number of elements in that row that are not NaN defines the number of vertices for that face.

The `Faces` and `Vertices` properties provide an alternative way to specify a patch that can be more efficient than using *x*, *y*, and *z* coordinates in most cases. For example, consider the following patch. It is composed of eight triangular faces defined by nine vertices.



F_1	V_1	V_4	V_5
F_2	V_1	V_5	V_2
F_3	V_2	V_5	V_6
F_4	V_2	V_6	V_3
F_5	V_4	V_7	V_8
F_6	V_4	V_8	V_5
F_7	V_5	V_8	V_9
F8	V_5	V_9	V_6

V_1	X_1	Y_1	Z_1
V_2	X_2	Y_2	Z_2
V_3	X_3	Y_3	Z_3
V_4	X_4	Y_4	Z_4
V_5	X_5	Y_5	Z_5
V_6	X_6	Y_6	Z_6
V_7	X_7	Y_7	Z_7
V_8	X_8	Y_8	Z_8
V_9	X_9	Y_9	Z_9

The corresponding `Faces` and `Vertices` properties are shown to the right of the patch. Note how some faces share vertices with other faces. For example, the fifth vertex (`V5`) is used six times, once each by faces one, two, and three and six, seven, and eight. Without sharing vertices, this same patch requires `24` vertex definitions.

`FaceVertexAlphaData`
m-by-1 matrix

Face and vertex transparency data. Specifies the transparency of patches that have been defined by the `Faces` and `Vertices` properties. The interpretation of the values specified for `FaceVertexAlphaData` depends on the dimensions of the data.

`FaceVertexAlphaData` can be one of the following:

- A single value, which applies the same transparency to the entire patch. The `FaceAlpha` property must be set to `flat`.
- An `m`-by-1 matrix (where `m` is the number of rows in the `Faces` property), which specifies one transparency value per face. The `FaceAlpha` property must be set to `flat`.
- An `m`-by-1 matrix (where `m` is the number of rows in the `Vertices` property), which specifies one transparency value per vertex. The `FaceAlpha` property must be set to `interp`.

The `AlphaDataMapping` property determines how MATLAB interprets the `FaceVertexAlphaData` property values.

`FaceVertexCData`
matrix

Face and vertex colors. Specifies the color of patches defined by the `Faces` and `Vertices` properties. You must also set the values of the `FaceColor`, `EdgeColor`, `MarkerFaceColor`, or `MarkerEdgeColor` appropriately. The interpretation of the values specified for `FaceVertexCData` depends on the dimensions of the data.

For indexed colors, `FaceVertexCData` can be:

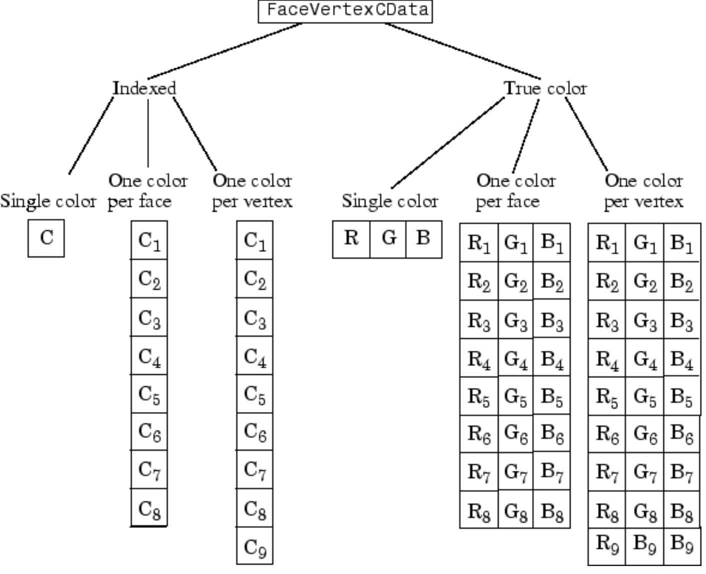
- A single value, which applies a single color to the entire patch.
- An *n*-by-1 matrix, where *n* is the number of rows in the `Faces` property, which specifies one color per face.

- An *n*-by-1 matrix, where *n* is the number of rows in the `Vertices` property, which specifies one color per vertex.

For true colors, `FaceVertexCData` can be:

- A 1-by-3 matrix, which applies a single color to the entire patch.
- An *n*-by-3 matrix, where *n* is the number of rows in the `Faces` property, which specifies one color per face.
- An *n*-by-3 matrix, where *n* is the number of rows in the `Vertices` property, which specifies one color per vertex.

The following diagram illustrates the various forms of the `FaceVertexCData` property for a patch having eight faces and nine vertices. The `CDataMapping` property determines how MATLAB interprets the `FaceVertexCData` property when you specify indexed colors.



`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

- `on` — Handles are always visible.
- `callback` — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

`HitTest`
`{on} | off`

Selectable by mouse click. Determines if the patch can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the patch. If `HitTest` is `off`, clicking the patch selects the object below it (which might be the axes containing it).

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the `Interruptible` property affects only the callbacks for the`ButtonDownFcn` property. A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both the callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.

- If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see "Control Callback Execution and Interruption".

`LineStyle`

`{-} | -- | : | -. | none`

Line style of the patch edges.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line

Specifier	Line Style
'-.'	Dash-dot line
'none'	No line

Use `LineStyle` `none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`
width in points

Edge line width. The width, in points, of the patch edges 1 point = $^1/_{72}$ inch. The default value is 0.5 points.

`Marker`
character (see table)

Marker symbol. Specifies marks that locate vertices. You can set values for the `Marker` property independently from the `LineStyle` property. For a list of supported marker symbols, see the following table.

Marker Specifiers Table

Specifier	Marker Type
'+'	Plus sign
'o'	Circle
'*'	Asterisk
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond

Specifier	Marker Type
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'''	Six-pointed star (hexagram)
'none'	No marker (default)

MarkerEdgeColor
`ColorSpec | none | {auto} | flat`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `ColorSpec` — Defines the color to use.
- `none` — Specifies no color, which makes nonfilled markers invisible.
- `auto` — Sets `MarkerEdgeColor` to the same color as the `EdgeColor` property.
- `flat` — The color of each vertex controls the color of the marker that denotes it.

MarkerFaceColor
`ColorSpec | {none} | auto | flat`

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `ColorSpec` — Defines the color to use.

- `none` — Makes the interior of the marker transparent, allowing the background to show through.
- `auto` — Sets the fill color to the axes color, or the figure color, if the axes `Color` property is `none`.
- `flat` — The color of each vertex controls the color of the marker that denotes it.

`MarkerSize`
scalar

Marker size. Size of the marker in points. The default value is `6`.

Note MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`NormalMode`
`{auto} | manual`

MATLAB generated or user-specified normal vectors.

- `auto` — MATLAB calculates vertex normals based on the coordinate data
- `manual` — If you specify your own vertex normals, MATLAB sets this property to `manual` and does not generate its own data.

See also the `VertexNormals` property.

`Parent`
handle of axes, hggroup, or hgtransform

Parent of patch object. Contains the handle of the patch object's parent. The parent of a patch object is the axes, hggroup, or hgtransform object that contains it. See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When this property is `on`, MATLAB displays selection handles or a dashed box (depending on the number of faces) if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by:

- Drawing handles at each vertex for a single-faced patch
- Drawing a dashed bounding box for a multifaced patch

When `SelectionHighlight` is `off`, MATLAB does not draw the indicators.

`SpecularColorReflectance`
scalar in the range `0` to `1`

Color of specularly-reflected light. When this property is `0`, the color of the specularly-reflected light depends on both the color of the object from which it reflects and the color of the light source. When set to `1`, the color of the specularly-reflected light depends only on the color or the light source (that is, the light object `Color` property). The proportions vary linearly for values in between.

`SpecularExponent`
scalar >= 1

Harshness of specular reflection. Controls the size of the specular spot. Most materials have exponents in the range of `5` to `20`.

`SpecularStrength`
scalar >= 0 and <= 1

Intensity of specular light. Sets the intensity of the specular component of the light falling on the patch. Specular light comes from light objects in the axes.

You can also set the intensity of the ambient and diffuse components of the light on the patch object. See the `AmbientStrength` and `DiffuseStrength` properties. Also see the `material` function.

`Tag`

string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

For example, suppose you use patch objects to create borders for a group of `uicontrol` objects and want to change the color of the borders in a uicontrol's callback routine. Specify a `Tag` with the patch definition:

```
patch(X,Y,'k','Tag','PatchBorder')
```

Then use `findobj` in the uicontrol's callback routine to obtain the handle of the patch and set its `FaceColor` property.

```
set(findobj('Tag','PatchBorder'),'FaceColor','w')
```

`Type`

string (read-only)

Class of the graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For patch objects, `Type` is always '`patch`'.

`UIContextMenu`
handle of uicontextmenu object

Associate a context menu with the patch. Assign this property the handle of a uicontextmenu object created in the same figure as the patch. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the patch.

`UserData`
matrix

User-specified data. Data you want to associate with the patch object. The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`VertexNormals`
matrix

Surface normal vectors. Contains the vertex normals for the patch. MATLAB generates this data to perform lighting calculations. You can supply your own vertex normal data, even if it does not match the coordinate data. Use this property to produce interesting lighting effects.

`Vertices`
matrix

Vertex coordinates. A matrix containing the *x*-, *y*-, *z*-coordinates for each vertex. See the `Faces` property for more information.

`Visible`
`{on} | off`

Patch object visibility.

- `on` — All patches are visible.
- `off` — The patch is not visible, but still exists, and you can query and set its properties.

`XData`
: vector | matrix

  X-coordinates. The *x*-coordinates of the patch vertices. If `XData` is a matrix, each column represents the *x*-coordinates of a single face of the patch. In this case, `XData`, `YData`, and `ZData` must have the same dimensions.

`YData`
: vector | matrix

  Y-coordinates. The *y*-coordinates of the patch vertices. If `YData` is a matrix, each column represents the *y*-coordinates of a single face of the patch. In this case, `XData`, `YData`, and `ZData` must have the same dimensions.

`ZData`
: vector | matrix

  Z-coordinates. The *z*-coordinates of the patch vertices. If `ZData` is a matrix, each column represents the *z*-coordinates of a single face of the patch. In this case, `XData`, `YData`, and `ZData` must have the same dimensions.

See Also

`patch`

path

Purpose

View or change search path

Alternatives

As an alternative to the `path` function, use the Set Path dialog box.

Syntax

```
path
path('newpath')
path(path,'newpath')
path('newpath',path)
p = path
```

Description

`path` displays the MATLAB search path, which is stored in `pathdef.m`.

`path('newpath')` changes the search path to `newpath`, where `newpath` is a string array of folders.

`path(path,'newpath')` adds the `newpath` folder to the end of the search path. If `newpath` is already on the search path, then `path(path, 'newpath')` moves `newpath` to the end of the search path.

`path('newpath',path)` adds the `newpath` folder to the top of the search path. If `newpath` is already on the search path, then `path('newpath', path)` moves `newpath` to the top of the search path. To add multiple folders in one statement, instead use `addpath`.

`p = path` returns the search path to string variable `p`.

Examples

Display the search path:

```
path
```

MATLAB returns, for example

```
MATLABPATH

 H:\My Documents\MATLAB
 C:\Program Files\MATLAB\R200nn\toolbox\matlab\general
 C:\Program Files\MATLAB\R200nn\toolbox\matlab\ops
 C:\Program Files\MATLAB\R200nn\toolbox\matlab\lang
 C:\Program Files\MATLAB\R200nn\toolbox\matlab\elmat
```

```
 C:\Program Files\MATLAB\R200nn\toolbox\matlab\elfun
...
```

`R200nn` represents the folder for the MATLAB release, for example, `R2009b`.

Add a new folder to the search path on Microsoft Windows platforms:

```
path(path,'c:/tools/goodstuff')
```

Add a new folder to the search path on UNIX[7] platforms:

```
path(path,'/home/tools/goodstuff')
```

Temporarily add the folder `my_files` to the search path, run `my_function` in `my_files`, then restore the previous search path:

```
p = path
path(p,'my_files')
my_function
path(p)
```

See Also

`addpath` | `cd` | `dir` | `genpath` | `matlabroot` | `pathsep` | `pathtool` | `rehash` | `restoredefaultpath` | `rmpath` | `savepath` | `startup` | `userpath` | `what`

How To

- "What Is the MATLAB Search Path?"
- "Files and Folders that MATLAB Accesses"

7. UNIX is a registered trademark of The Open Group in the United States and other countries.

Purpose

Save current search path to `pathdef.m` file

Syntax

```
path2rc
```

Description

`path2rc` runs `savepath`. The `savepath` function is replacing `path2rc`. Use `savepath` instead of `path2rc` and replace instances of `path2rc` with `savepath`.

pathsep

Purpose

Search path separator for current platform

Syntax

```
c = pathsep
```

Description

`c = pathsep` returns the search path separator character for this platform. The search path separator is the character that separates path names in the `pathdef.m` file, as returned by the `path` function. The character is a semicolon (`;`). For versions of MATLAB software earlier than version 7.7 (R2008b), the character on UNIX[8] platforms was a colon (`:`). Use `pathsep` to work programmatically with the content of the search path file.

See Also

`fileparts` | `filesep` | `fullfile` | `path`

How To

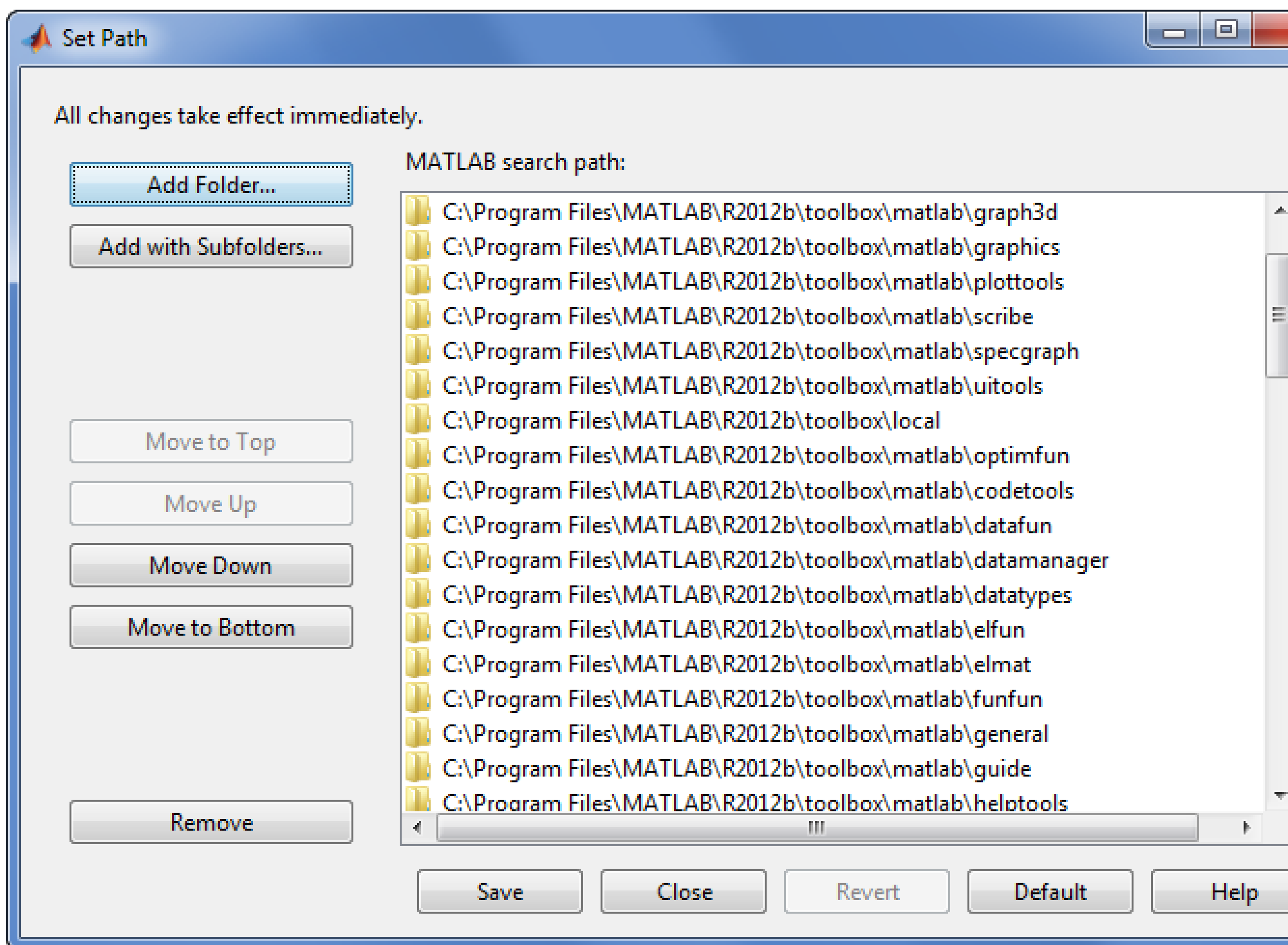
- "What Is the MATLAB Search Path?"

8. UNIX is a registered trademark of The Open Group in the United States and other countries.

Purpose Open Set Path dialog box to view and change search path

Syntax `pathtool`

Description `pathtool` opens the Set Path dialog box, a graphical user interface you use to view and modify the MATLAB search path.



See Also

`addpath` | `cd` | `dir` | `genpath` | `matlabroot` | `path` | `pathsep` | `rehash` | `restoredefaultpath` | `rmpath` | `savepath` | `startup` | `what`

How To

- “What Is the MATLAB Search Path?”

Purpose

Halt execution temporarily

Syntax

```
pause
pause(n)
pause on
pause off
pause query
state = pause('query')
oldstate = pause(newstate)
```

Description

`pause`, by itself, causes the currently executing function to stop and wait for you to press any key before continuing. Pausing must be enabled for this to take effect. (See `pause on`, below). `pause` without arguments also blocks execution of Simulink models, but not repainting of them.

`pause(n)` pauses execution for `n` seconds before continuing, where `n` is any nonnegative real number. Pausing must be enabled for this to take effect.

Typing `pause(inf)` puts you into an infinite loop. To return to the MATLAB prompt, type **Ctrl+C**.

`pause on` enables the pausing of MATLAB execution via the `pause` and `pause(n)` commands. Pausing remains enabled until you enter `pause off` in your function or at the command line.

`pause off` disables the pausing of MATLAB execution via the `pause` and `pause(n)` commands. This allows normally interactive scripts to run unattended. Pausing remains disabled until you enter `pause on` in your function or at the command line, or start a new MATLAB session.

`pause query` displays `'on'` if pausing is currently enabled. Otherwise, it displays `'off'`.

`state = pause('query')` returns `'on'` in character array `state` if pausing is currently enabled. Otherwise, the value of `state` is `'off'`.

`oldstate = pause(newstate)`, enables or disables pausing, depending on the `'on'` or `'off'` value in `newstate`, and returns the former setting (also either `'on'` or `'off'`) in character array `oldstate`.

Tips

The accuracy of `pause` is subject to the scheduling resolution of the operating system you are using, and also to other system activity. It cannot be guaranteed with 100% confidence. Asking for finer resolutions shows higher relative error.

While MATLAB is paused, the following continue to execute:

- Repainting of figure windows, Simulink block diagrams, and Java windows
- HG callbacks from figure windows
- Event handling from Java windows

See Also

`keyboard` | `input` | `drawnow`

Purpose

Set or query plot box aspect ratio

Syntax

```
pbaspect
pbaspect([aspect_ratio])
pbaspect('mode')
pbaspect('auto')
pbaspect('manual')
pbaspect(axes_handle,...)
```

Description

The plot box aspect ratio determines the relative size of the *x*-, *y*-, and *z*-axes.

`pbaspect` with no arguments returns the plot box aspect ratio of the current axes.

`pbaspect([aspect_ratio])` sets the plot box aspect ratio in the current axes to the specified value. Specify the aspect ratio as three relative values representing the ratio of the *x*-, *y*-, and *z*-axes size. For example, a value of `[1 1 1]` (the default) means the plot box is a cube (although with stretch-to-fill enabled, it may not appear as a cube). See Tips.

`pbaspect('mode')` returns the current value of the plot box aspect ratio mode, which can be either `auto` (the default) or `manual`. See Remarks.

`pbaspect('auto')` sets the plot box aspect ratio mode to `auto`.

`pbaspect('manual')` sets the plot box aspect ratio mode to `manual`.

`pbaspect(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. If you do not specify an axes handle, `pbaspect` operates on the current axes.

Tips

`pbaspect` sets or queries values of the axes object `PlotBoxAspectRatio` and `PlotBoxAspectRatioMode` properties.

When the plot box aspect ratio mode is `auto`, the MATLAB software sets the ratio to `[1 1 1]`, but may change it to accommodate manual settings of the data aspect ratio, camera view angle, or axis limits. See the axes `DataAspectRatio` property for a table listing the interactions between various properties.

Setting a value for the plot box aspect ratio or setting the plot box aspect ratio mode to `manual` disables the MATLAB stretch-to-fill feature (stretching of the axes to fit the window). This means setting the plot box aspect ratio to its current value,

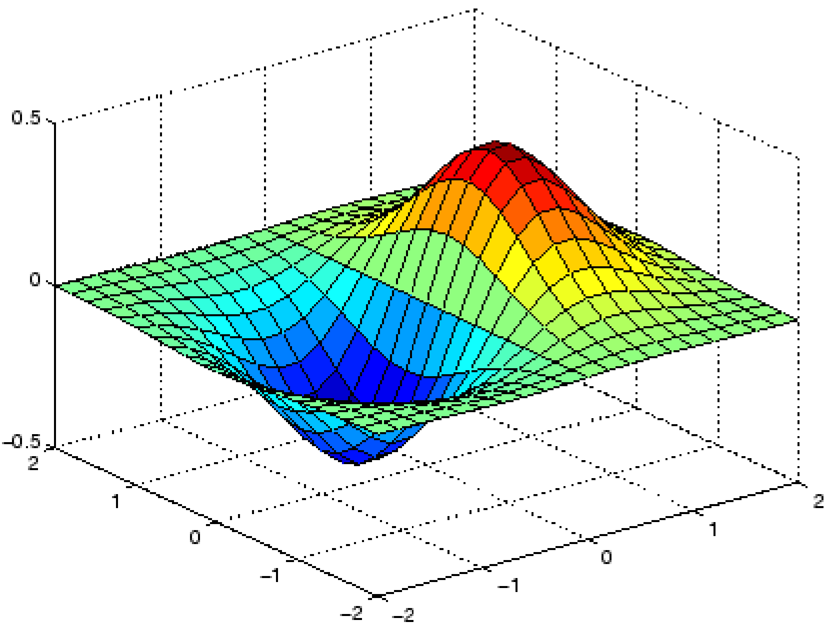
```
pbaspect(pbaspect)
```

can cause a change in the way the graphs look. See the Remarks section of the `axes` reference description, "Axes Aspect Ratio Properties" in the 3-D Visualization manual, and "Setting Aspect Ratio" in the MATLAB Graphics manual for a discussion of stretch-to-fill.

Examples

The following surface plot of the function $z = xe^{(-x^2 - y^2)}$ is useful to illustrate the plot box aspect ratio. First plot the function over the range $-2 \le x \le 2$, $-2 \le y \le 2$,

```
[x,y] = meshgrid([-2:.2:2]);
z = x.*exp(-x.^2 - y.^2);
surf(x,y,z)
```



Querying the plot box aspect ratio shows that the plot box is square.

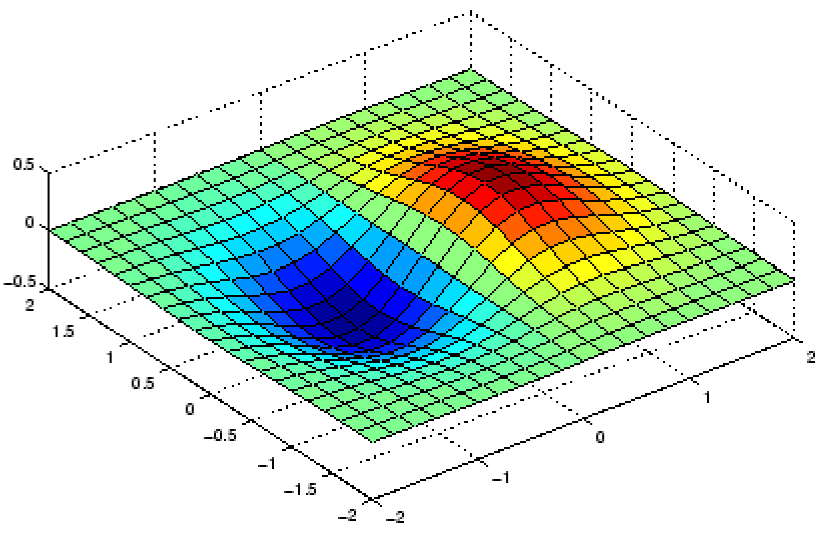
```
pbaspect
ans =
     1   1   1
```

It is also interesting to look at the data aspect ratio selected by MATLAB.

```
daspect
ans =
     4   4   1
```

To illustrate the interaction between the plot box and data aspect ratios, set the data aspect ratio to `[1 1 1]` and again query the plot box aspect ratio.

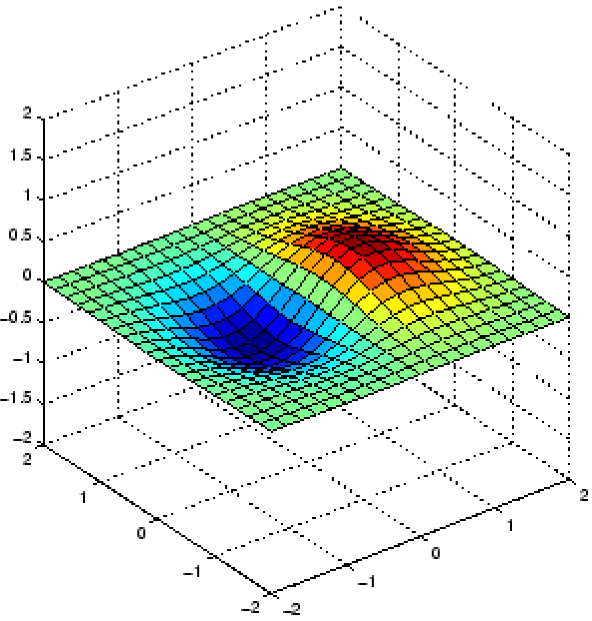
```
daspect([1 1 1])
```




```
pbaspect
ans =
     4   4   1
```

The plot box aspect ratio has changed to accommodate the specified data aspect ratio. Now suppose you want the plot box aspect ratio to be [1 1 1] as well.

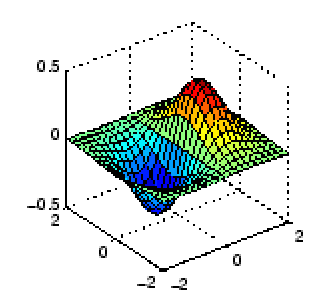
```
pbaspect([1 1 1])
```

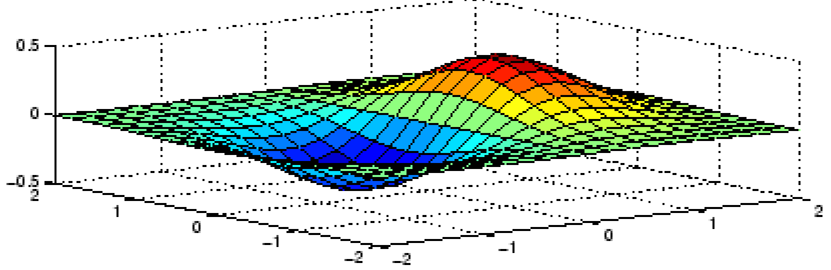


Notice how MATLAB changed the axes limits because of the constraints introduced by specifying both the plot box and data aspect ratios.

You can also use `pbaspect` to disable stretch-to-fill. For example, displaying two subplots in one figure can give surface plots a squashed appearance. Disabling stretch-to-fill,

```
upper_plot = subplot(211);
surf(x,y,z)
lower_plot = subplot(212);
surf(x,y,z)
pbaspect(upper_plot,'manual')
```





See Also

`axis` | `daspect` | `xlim` | `ylim` | `zlim` | `DataAspectRatio` | `PlotBoxAspectRatio` | `XLim` | `YLim` | `ZLim`

How To

- Setting Aspect Ratio
- Axes Aspect Ratio Properties

Purpose

Preconditioned conjugate gradients method

Syntax

```
x = pcg(A,b)
pcg(A,b,tol)
pcg(A,b,tol,maxit)
pcg(A,b,tol,maxit,M)
pcg(A,b,tol,maxit,M1,M2)
pcg(A,b,tol,maxit,M1,M2,x0)
[x,flag] = pcg(A,b,...)
[x,flag,relres] = pcg(A,b,...)
[x,flag,relres,iter] = pcg(A,b,...)
[x,flag,relres,iter,resvec] = pcg(A,b,...)
```

Description

`x = pcg(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be symmetric and positive definite, and should also be large and sparse. The column vector `b` must have length `n`. You also can specify `A` to be a function handle, `afun`, such that `afun(x)` returns `A*x`.

"Parameterizing Functions" explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `pcg` converges, a message to that effect is displayed. If `pcg` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`pcg(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `pcg` uses the default, `1e-6`.

`pcg(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `pcg` uses the default, `min(n,20)`.

`pcg(A,b,tol,maxit,M)` and `pcg(A,b,tol,maxit,M1,M2)` use symmetric positive definite preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]`

then pcg applies no preconditioner. M can be a function handle mfun such that mfun(x) returns M\x.

pcg(A,b,tol,maxit,M1,M2,x0) specifies the initial guess. If x0 is [], then pcg uses the default, an all-zero vector.

[x,flag] = pcg(A,b,...) also returns a convergence flag.

Flag	Convergence
0	pcg converged to the desired tolerance tol within maxit iterations.
1	pcg iterated maxit times but did not converge.
2	Preconditioner M was ill-conditioned.
3	pcg stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during pcg became too small or too large to continue computing.

Whenever flag is not 0, the solution x returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the flag output is specified.

[x,flag,relres] = pcg(A,b,...) also returns the relative residual norm(b-A*x)/norm(b). If flag is 0, relres <= tol.

[x,flag,relres,iter] = pcg(A,b,...) also returns the iteration number at which x was computed, where 0 <= iter <= maxit.

[x,flag,relres,iter,resvec] = pcg(A,b,...) also returns a vector of the residual norms at each iteration including norm(b-A*x0).

Examples

Using pcg with Large Matrices

This example shows how to use pcg with a matrix input and with a function handle.

```
n1 = 21;
A = gallery('moler',n1);
b1 = sum(A,2);
```

```
tol = 1e-6;
maxit = 15;
M1 = spdiags((1:n1)',0,n1,n1);
[x1,flag1,rr1,iter1,rv1] = pcg(A,b1,tol,maxit,M1);
```

Alternatively, you can use the following function in place of the matrix `A`:

```
function y = applyMoler(x)
y = x;
y(end-1:-1:1) = y(end-1:-1:1) - cumsum(y(end:-1:2));
y(2:end) = y(2:end) - cumsum(y(1:end-1));
```

By using this function, you can solve larger systems more efficiently as there is no need to store the entire matrix `A`:

```
n2 = 21;
b2 = applyMoler(ones(n2,1));
tol = 1e-6;
maxit = 15;
M2 = spdiags((1:n2)',0,n2,n2);
[x2,flag2,rr2,iter2,rv2] = pcg(@applyMoler,b2,tol,maxit,M2);
```

Using pcg with a Preconditioner

This example demonstrates how to use a preconditioner matrix with `pcg`.

1

Create an input matrix and try to solve the system with `pcg`:

```
A = delsq(numgrid('S',100));
b = ones(size(A,1),1);
[x0,fl0,rr0,it0,rv0] = pcg(A,b,1e-8,100);
```

`fl0` is `1` because `pcg` does not converge to the requested tolerance of `1e-8` within the requested maximum 100 iterations. A preconditioner can make the system converge more quickly.

2

Use `ichol` with only one input argument to construct an incomplete Cholesky factorization with zero fill:

```
L = ichol(A);
[x1,fl1,rr1,it1,rv1] = pcg(A,b,1e-8,100,L,L');
```

`fl1` is `0` because `pcg` drives the relative residual to `9.8e-09` (the value of `rr1`) which is less than the requested tolerance of `1e-8` at the seventy-seventh iteration (the value of `it1`) when preconditioned by the zero-fill incomplete Cholesky factorization. `rv1(1) = norm(b)` and `rv1(78) = norm(b-A*x1)`.

The previous matrix represents the discretization of the Laplacian on a 100x100 grid with Dirichlet boundary conditions. This means that a modified incomplete Cholesky preconditioner might perform even better.

3

Use the `michol` option to create a modified incomplete Cholesky preconditioner:

```
L = ichol(A,struct(`michol','on'));
[x2,fl2,rr2,it2,rv2] = pcg(A,b,1e-8,100,L,L');
```

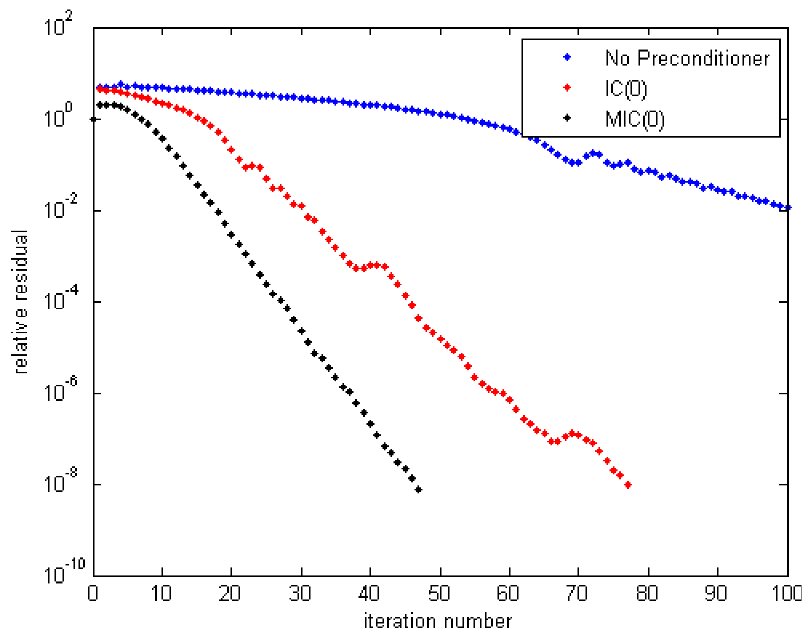
In this case you attain convergence in only forty-seven iterations.

4

You can see how the preconditioners affect the rate of convergence of `pcg` by plotting each of the residual histories starting from the initial estimate (iterate number `0`):

```
figure;
semilogy(0:it0,rv0/norm(b),'b.');
hold on;
semilogy(0:it1,rv1/norm(b),'r.');
semilogy(0:it2,rv2/norm(b),'k.');
legend('No Preconditioner','IC(0)','MIC(0)');
xlabel('iteration number');
```

```
ylabel('relative residual');
hold off;
```



References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

See Also

`bicg` | `bicgstab` | `cgs` | `cholinc` | `gmres` | `lsqr` | `minres` | `qmr` | `symmlq` | `function_handle` | `mldivide`

pchip

Purpose

Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)

Syntax

```
yi = pchip(x,y,xi)
pp = pchip(x,y)
```

Description

`yi = pchip(x,y,xi)` returns vector `yi` containing elements corresponding to the elements of `xi` and determined by piecewise cubic interpolation within vectors `x` and `y`. The vector `x` specifies the points at which the data `y` is given. If `y` is a matrix, then the interpolation is performed for each column of `y` and `yi` is `length(xi)`-by-`size(y,2)`.

`pp = pchip(x,y)` returns a piecewise polynomial structure for use by `ppval`. `x` can be a row or column vector. `y` is a row or column vector of the same length as `x`, or a matrix with `length(x)` columns.

`pchip` finds values of an underlying interpolating function $P(x)$ at intermediate points, such that:

- On each subinterval $x_k \le x \le x_{k+1}$, $P(x)$ is the cubic Hermite interpolant to the given values and certain slopes at the two endpoints.
- $P(x)$ interpolates y, i.e., $P(x_j) = y_j$, and the first derivative $P'(x)$ is continuous. $P''(x)$ is probably not continuous; there may be jumps at the x_j.
- The slopes at the x_j are chosen in such a way that $P(x)$ preserves the shape of the data and respects monotonicity. This means that, on intervals where the data are monotonic, so is $P(x)$; at points where the data has a local extremum, so does $P(x)$.

Note If y is a matrix, $P(x)$ satisfies the above for each column of y.

Tips

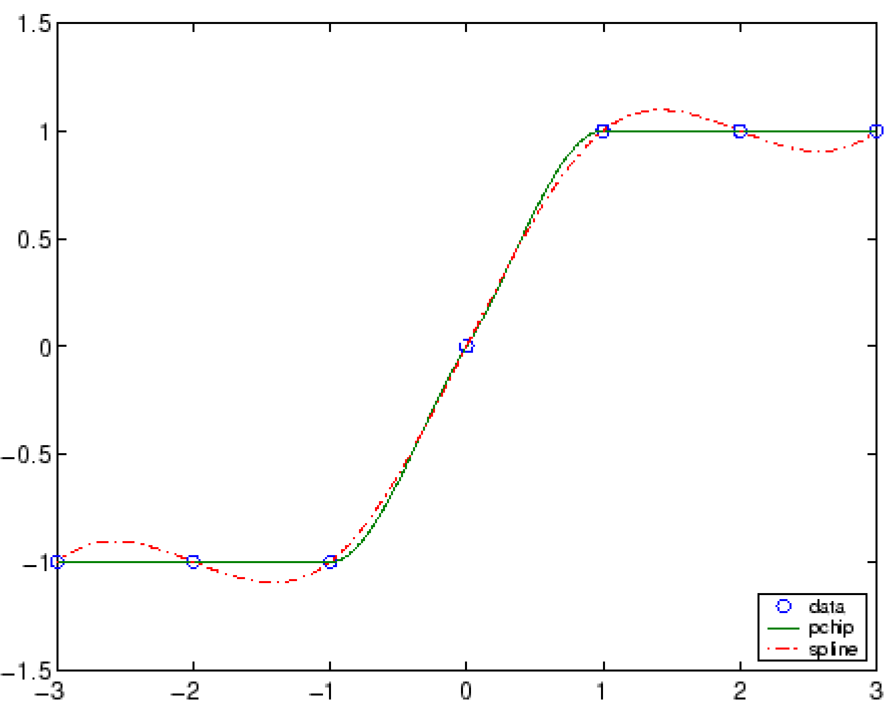
`spline` constructs $S(x)$ in almost the same way `pchip` constructs $P(x)$.

However, `spline` chooses the slopes at the x_j differently, namely to make even $S''(x)$ continuous. This has the following effects:

- `spline` produces a smoother result, i.e. $S''(x)$ is continuous.
- `spline` produces a more accurate result if the data consists of values of a smooth function.
- `pchip` has no overshoots and less oscillation if the data are not smooth.
- `pchip` is less expensive to set up.
- The two are equally expensive to evaluate.

Examples

```
x = -3:3;
y = [-1 -1 -1 0 1 1 1];
t = -3:.01:3;
p = pchip(x,y,t);
s = spline(x,y,t);
plot(x,y,'o',t,p,'-',t,s,'-.')
legend('data','pchip','spline',4)
```



References

[1] Fritsch, F. N. and R. E. Carlson, "Monotone Piecewise Cubic Interpolation," *SIAM J. Numerical Analysis*, Vol. 17, 1980, pp.238-246.

[2] Kahaner, David, Cleve Moler, Stephen Nash, *Numerical Methods and Software*, Prentice Hall, 1988.

See Also

`interp1` | `spline` | `ppval`

Purpose

Create protected function file

Syntax

```
pcode(fun)
pcode(fun1,...,funN)
pcode(fun,'-inplace')
```

Description

`pcode(fun)` encrypts the code in `fun.m` and produces a file called `fun.p`, known as a P-file. If `fun` is a folder, then all the script or function files in that folder are encrypted in P-files. MATLAB creates the P-files in the current folder. The original `.m` file or folder can be anywhere on the search path.

`pcode(fun1,...,funN)` creates `N` P-files from the listed files. If any inputs are folders, then MATLAB creates a P-file for every `.m` file the folders contain.

`pcode(fun,'-inplace')` creates P-files in the same folder as the script or function files.

Tips

- The `pcode` algorithm was redesigned in MATLAB 7.5 (Release R2007b). You can run older P-files in any current version of MATLAB; however, upcoming releases will not run P-files created before version 7.5. Files generated in 7.5, or later versions, cannot run in MATLAB 7.4 or earlier.
- When encrypting all files in a folder, `pcode` does not encrypt any files within subfolders.
- A P-file takes precedence over the corresponding `.m` file for execution, even after modifications to the `.m` file.
- MATLAB does not display any of the help comments that might be in the original `.m` file.

Input Arguments

fun

MATLAB file or directory containing MATLAB files. If `fun` resides within a package and/or class folder, then `pcode` creates the same package and/or class structure to house the resulting P-files.

An input argument with no file extension and that is not a folder must be a function in the MATLAB path or in the current folder.

When using wild cards `*`, `pcode` ignores all files with extensions other than `.m`.

Examples

P-Coding Multiple Files

Convert selected files from the `sparfun` folder into P-files.

Create a temporary folder and define an existing path to `.m` files.

```
tmp = tempname;
mkdir(tmp);
cd(tmp);
fun = fullfile(matlabroot,'toolbox','matlab','sparfun','spr*.m');
```

Create the P-files.

```
pcode(fun)
dir(tmp)
```

```
.            ..           sprand.p     sprandn.p    sprandsym.p  sprank.p
```

The temporary folder now contains encoded P-files.

P-Coding Files That Belong to a Package and/or Class

Generate P-files from input files that are part of a package and/or class.

Define `funclass` as an existing a class folder that contains `.m` files.

```
funclass = fullfile(matlabroot,'toolbox','matlab','datatypes','@char')
dir(funclass)
```

```
funclass =

C:\Program Files\MATLAB\R2011b\toolbox\matlab\datatypes\@char
```

```
.                    ..                   superiorfloat.m
```

Create a temporary folder. This folder has no package or class structure at this time.

```
tmp = tempname;
mkdir(tmp);
cd(tmp);
dir(tmp)
```

```
.                ..
```

Create a P-file for every `.m` file in the path `funclass`. Because the input files are part of a package and/or class, MATLAB creates a folder structure so that the output file belongs to the same package and/or class.

```
pcode(funclass)
dir(tmp)
```

```
.       ..      @char
```

You see that the P-file resides in the same folder structure.

```
dir('@char')
```

```
.                    ..                   superiorfloat.p
```

P-Coding In Place

Generate P-files in the same folder as the input files using the option `inplace`

Copy several MATLAB files to a temporary folder.

```
fun = fullfile(matlabroot,'toolbox','matlab','sparfun','spr*.m');
tmp = tempname;
mkdir(tmp);
copyfile(fun,tmp)
dir(tmp)
```

```
.             ..            sprand.m      sprandn.m     sprandsym.m   sprank.m
```

Create P-files in the same folder as the original `.m` files.

```
pcode(tmp,'-inplace')
dir(tmp)
```

```
.             sprand.m      sprandn.m     sprandsym.m   sprank.m
..            sprand.p      sprandn.p     sprandsym.p   sprank.p
```

See Also

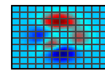
`depdir` | `depfun`

Concepts

- "Protect Your Source Code"

Purpose

Pseudocolor (checkerboard) plot



Syntax

```
pcolor(C)
pcolor(X,Y,C)
pcolor(axes_handles,...)
h = pcolor(...)
```

Description

A pseudocolor plot is a rectangular array of cells with colors determined by `C`. MATLAB creates a pseudocolor plot using each set of four adjacent points in `C` to define a surface rectangle (i.e., cell).

The default `shading` is `faceted`, which colors each cell with a single color. The last row and column of `C` are not used in this case. With `shading interp`, each cell is colored by bilinear interpolation of the colors at its four vertices, using all elements of `C`.

The minimum and maximum elements of `C` are assigned the first and last colors in the colormap. Colors for the remaining elements in `C` are determined by a linear mapping from value to colormap element.

`pcolor(C)` draws a pseudocolor plot. The elements of `C` are linearly mapped to an index into the current colormap. The mapping from `C` to the current colormap is defined by `colormap` and `caxis`.

`pcolor(X,Y,C)` draws a pseudocolor plot of the elements of `C` at the locations specified by `X` and `Y`. The plot is a logically rectangular, two-dimensional grid with vertices at the points `[X(i,j), Y(i,j)]`. `X` and `Y` are vectors or matrices that specify the spacing of the grid lines. If `X` and `Y` are vectors, `X` corresponds to the columns of `C` and `Y` corresponds to the rows. If `X` and `Y` are matrices, they must be the same size as `C`.

`pcolor(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = pcolor(...)` returns a handle to a `surface` graphics object.

Tips

A pseudocolor plot is a flat surface plot viewed from above. `pcolor(X,Y,C)` is the same as viewing `surf(X,Y,zeros(size(X)),C)` using `view([0 90])`.

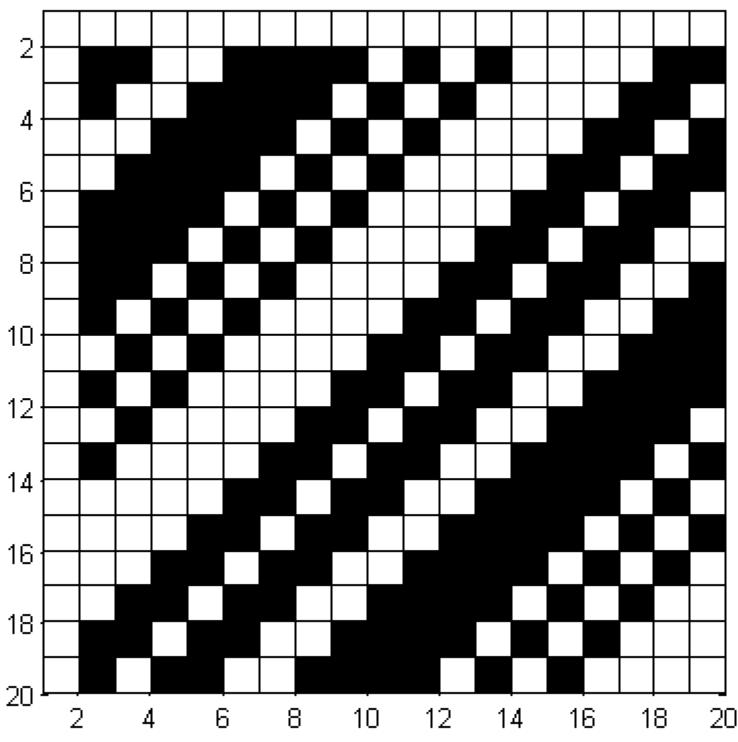
When you use `shading faceted` or `shading flat`, the constant color of each cell is the color associated with the corner having the smallest *x*-*y* coordinates. Therefore, `C(i,j)` determines the color of the cell in the *i*th row and *j*th column. The last row and column of `C` are not used.

When you use `shading interp`, each cell's color results from a bilinear interpolation of the colors at its four vertices, and all elements of `C` are used.

Examples

A Hadamard matrix has elements that are `+1` and `-1`. A colormap with only two entries is appropriate when displaying a pseudocolor plot of this matrix.

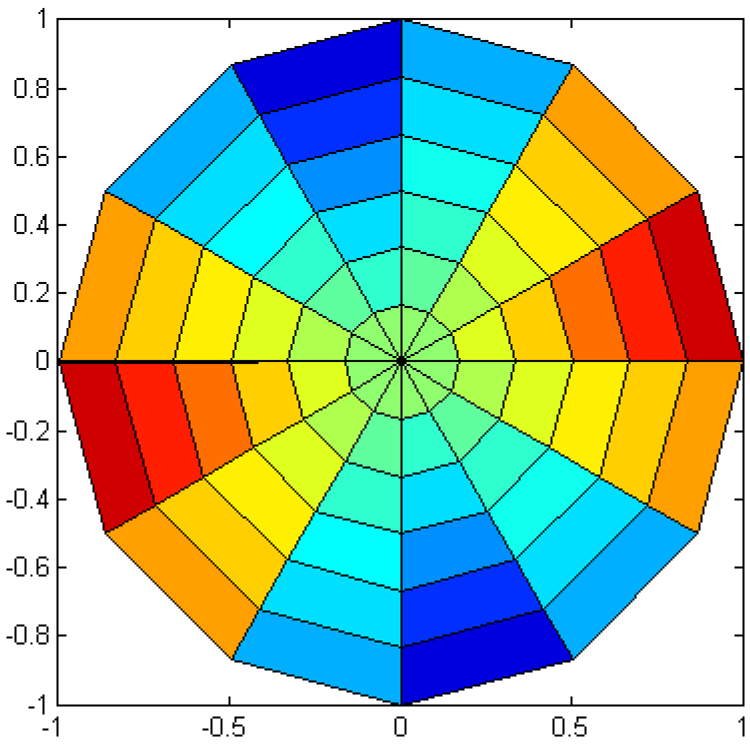
```
pcolor(hadamard(20))
colormap(gray(2))
axis ij
axis square
```



A simple color wheel illustrates a polar coordinate system.

```
n = 6;
r = (0:n)'/n;
theta = pi*(-n:n)/n;
X = r*cos(theta);
Y = r*sin(theta);
C = r*cos(2*theta);
pcolor(X,Y,C)
```

```
axis equal tight
```



Algorithms

The number of vertex colors for `pcolor(C)` is the same as the number of cells for `image(C)`. `pcolor` differs from `image` in that `pcolor(C)` specifies the colors of vertices, which are scaled to fit the colormap; changing the axes `clim` property changes this color mapping. `image(C)` specifies the colors of cells and directly indexes into the colormap without scaling. Additionally, `pcolor(X,Y,C)` can produce parametric grids, which is not possible with `image`.

See Also

`caxis` | `image` | `mesh` | `shading` | `surf` | `view`

pdepe

Purpose

Solve initial-boundary value problems for parabolic-elliptic PDEs in 1-D

Syntax

```
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)
[sol,tsol,sole,te,ie] = pdepe(m,pdefun,icfun,bcfun,xmesh,
   tspan,options)
```

Arguments

m	A parameter corresponding to the symmetry of the problem. m can be slab = 0, cylindrical = 1, or spherical = 2.
pdefun	A handle to a function that defines the components of the PDE.
icfun	A handle to a function that defines the initial conditions.
bcfun	A handle to a function that defines the boundary conditions.
xmesh	A vector [x0, x1, ..., xn] specifying the points at which a numerical solution is requested for every value in tspan. The elements of xmesh must satisfy x0 < x1 < ... < xn. The length of xmesh must be >= 3.
tspan	A vector [t0, t1, ..., tf] specifying the points at which a solution is requested for every value in xmesh. The elements of tspan must satisfy t0 < t1 < ... < tf. The length of tspan must be >= 3.
options	Some options of the underlying ODE solver are available in pdepe: RelTol, AbsTol, NormControl, InitialStep, MaxStep, and Events. In most cases, default values for these options provide satisfactory solutions. See odeset for details.

Description

`sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)` solves initial-boundary value problems for systems of parabolic and elliptic PDEs in the one space variable x and time t. `pdefun`, `icfun`, and `bcfun` are function handles. See the `function_handle` reference page for more information. The ordinary differential equations (ODEs) resulting from discretization in space are integrated to obtain approximate solutions at times specified in `tspan`. The `pdepe` function returns values of the solution on a mesh provided in `xmesh`.

"Parameterizing Functions" explains how to provide additional parameters to the functions `pdefun`, `icfun`, or `bcfun`, if necessary.

`pdepe` solves PDEs of the form:

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m f\left(x,t,u,\frac{\partial u}{\partial x}\right)\right) + s\left(x,t,u,\frac{\partial u}{\partial x}\right) \quad \textbf{(1-4)}$$

The PDEs hold for $t_0 \le t \le t_f$ and $a \le x \le b$. The interval $[a,b]$ must be finite. m can be 0, 1, or 2, corresponding to slab, cylindrical, or spherical symmetry, respectively. If $m > 0$, then a must be ≥ 0.

In Equation 1-4, $f\,(x,t,u,\partial u/\partial x)$ is a flux term and $s\,(x,t,u,\partial u/\partial x)$ is a source term. The coupling of the partial derivatives with respect to time is restricted to multiplication by a diagonal matrix $c\,(x,t,u,\partial u/\partial x)$. The diagonal elements of this matrix are either identically zero or positive. An element that is identically zero corresponds to an elliptic equation and otherwise to a parabolic equation. There must be at least one parabolic equation. An element of c that corresponds to a parabolic equation can vanish at isolated values of x if those values of x are mesh points. Discontinuities in c and/or s due to material interfaces are permitted provided that a mesh point is placed at each interface.

For $t = t_0$ and all x, the solution components satisfy initial conditions of the form

$$u(x,t_0) = u_0(x) \quad \textbf{(1-5)}$$

For all t and either $x = a$ or $x = b$, the solution components satisfy a boundary condition of the form

$$p(x,t,u) + q(x,t) f\left(x,t,u,\frac{\partial u}{\partial x}\right) = 0 \tag{1-6}$$

Elements of q are either identically zero or never zero. Note that the boundary conditions are expressed in terms of the flux f rather than $\partial u/\partial x$. Also, of the two coefficients, only p can depend on u.

In the call `sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)`:

- `m` corresponds to m.
- `xmesh(1)` and `xmesh(end)` correspond to a and b.
- `tspan(1)` and `tspan(end)` correspond to t_0 and t_f.
- `pdefun` computes the terms c, f, and s (Equation 1-4). It has the form

  ```
  [c,f,s] = pdefun(x,t,u,dudx)
  ```

  The input arguments are scalars `x` and `t` and vectors `u` and `dudx` that approximate the solution u and its partial derivative with respect to x, respectively. `c`, `f`, and `s` are column vectors. `c` stores the diagonal elements of the matrix c (Equation 1-4).
- `icfun` evaluates the initial conditions. It has the form

  ```
  u = icfun(x)
  ```

  When called with an argument `x`, `icfun` evaluates and returns the initial values of the solution components at `x` in the column vector `u`.
- `bcfun` evaluates the terms p and q of the boundary conditions (Equation 1-6). It has the form

  ```
  [pl,ql,pr,qr] = bcfun(xl,ul,xr,ur,t)
  ```

  `ul` is the approximate solution at the left boundary `xl` = a and `ur` is the approximate solution at the right boundary `xr` = b. `pl` and `ql` are column vectors corresponding to p and q evaluated at `xl`, similarly `pr` and `qr` correspond to `xr`. When $m > 0$ and $a = 0$, boundedness of the solution near $x = 0$ requires that the flux f vanish at $a = 0$. `pdepe`

imposes this boundary condition automatically and it ignores values returned in `pl` and `ql`.

`pdepe` returns the solution as a multidimensional array `sol`. u_i = `ui` = `sol(:,:,i)` is an approximation to the `i`th component of the solution vector u. The element `ui(j,k)` = `sol(j,k,i)` approximates u_i at (t,x) = (`tspan(j)`,`xmesh(k)`).

`ui = sol(j,:,i)` approximates component `i` of the solution at time `tspan(j)` and mesh points `xmesh(:)`. Use `pdeval` to compute the approximation and its partial derivative $\partial u_i/\partial x$ at points not included in `xmesh`. See `pdeval` for details.

`sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)` solves as above with default integration parameters replaced by values in `options`, an argument created with the `odeset` function. Only some of the options of the underlying ODE solver are available in `pdepe`: `RelTol`, `AbsTol`, `NormControl`, `InitialStep`, and `MaxStep`. The defaults obtained by leaving off the input argument `options` will generally be satisfactory. See `odeset` for details.

`[sol,tsol,sole,te,ie] = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)` with the `'Events'` property in `options` set to a function handle `Events`, solves as above while also finding where event functions `g(t,u(x,t))`are zero. For each function you specify whether the integration is to terminate at a zero and whether the direction of the zero crossing matters. Three column vectors are returned by `events`: `[value,isterminal,direction] = events(m,t,xmesh,umesh)`. `xmesh` contains the spatial mesh and `umesh` is the solution at the mesh points. Use `pdeval` to evaluate the solution between mesh points. For the I-th event function, `value(i)` is the value of the function, `ISTERMINAL(I) = 1` if the integration is to terminate at a zero of this event function and 0 otherwise. `direction(i) = 0` if all zeros are to be computed (the default), +1 if only zeros where the event function is increasing, and -1 if only zeros where the event function is decreasing. Output `tsol` is a column vector of times specified in `tspan`, prior to first terminal event. `SOL(j,:,:)` is the solution at `T(j)`. `TE` is a vector of

times at which events occur. `SOLE(j,:,:)` is the solution at `TE(j)` and indices in vector `IE` specify which event occurred.

If `UI = SOL(j,:,i)` approximates component i of the solution at time `TSPAN(j)` and mesh points `XMESH`, `pdeval` evaluates the approximation and its partial derivative $\partial u_i/\partial x$ at the array of points `XOUT` and returns them in `UOUT` and `DUOUTDX`: `[UOUT,DUOUTDX] = PDEVAL(M,XMESH,UI,XOUT)`

Note The partial derivative $\partial u_i/\partial x$ is evaluated here rather than the flux. The flux is continuous, but at a material interface the partial derivative may have a jump.

Tips

- The arrays `xmesh` and `tspan` play different roles in `pdepe`.

  `tspan` – The `pdepe` function performs the time integration with an ODE solver that selects both the time step and formula dynamically. The elements of `tspan` merely specify where you want answers and the cost depends weakly on the length of `tspan`.

  `xmesh` – Second order approximations to the solution are made on the mesh specified in `xmesh`. Generally, it is best to use closely spaced mesh points where the solution changes rapidly. `pdepe` does *not* select the mesh in x automatically. You must provide an appropriate fixed mesh in `xmesh`. The cost depends strongly on the length of `xmesh`. When $m > 0$, it is not necessary to use a fine mesh near $x = 0$ to account for the coordinate singularity.

- The time integration is done with `ode15s`. `pdepe` exploits the capabilities of `ode15s` for solving the differential-algebraic equations that arise when Equation 1-4 contains elliptic equations, and for handling Jacobians with a specified sparsity pattern.

- After discretization, elliptic equations give rise to algebraic equations. If the elements of the initial conditions vector that correspond to elliptic equations are not "consistent" with the discretization, `pdepe` tries to adjust them before beginning the time integration. For

this reason, the solution returned for the initial time may have a discretization error comparable to that at any other time. If the mesh is sufficiently fine, `pdepe` can find consistent initial conditions close to the given ones. If `pdepe` displays a message that it has difficulty finding consistent initial conditions, try refining the mesh.

No adjustment is necessary for elements of the initial conditions vector that correspond to parabolic equations.

Examples

Example 1. This example illustrates the straightforward formulation, computation, and plotting of the solution of a single PDE.

$$\pi^2 \frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}\right)$$

This equation holds on an interval $0 \le x \le 1$ for times $t \ge 0$.

The PDE satisfies the initial condition

$$u(x,0) = \sin \pi x$$

and boundary conditions

$$u(0,t) \equiv 0$$

$$\pi e^{-t} + \frac{\partial u}{\partial x}(1,t) = 0$$

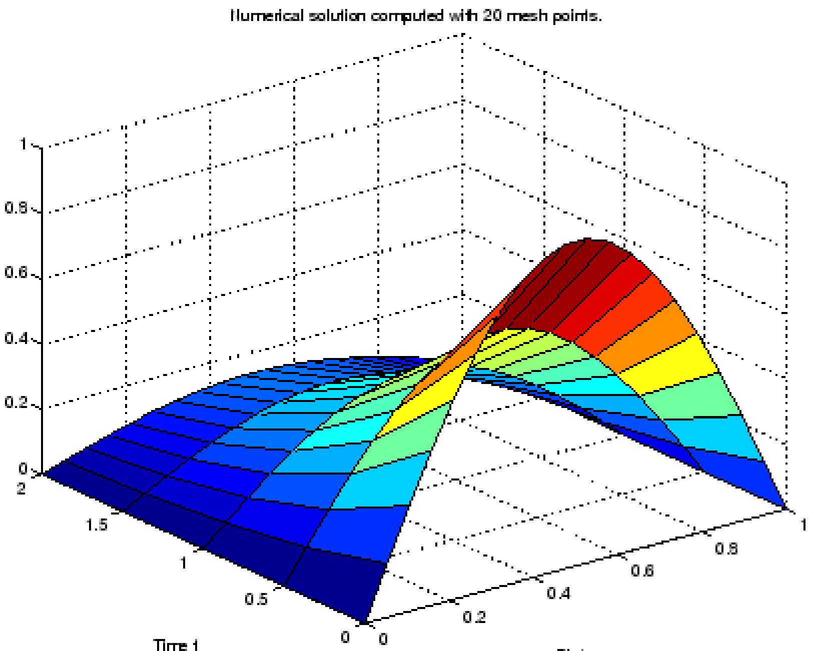
It is convenient to use local functions to place all the functions required by `pdepe` in a single function.

```
function pdex1

m = 0;
x = linspace(0,1,20);
t = linspace(0,2,5);

sol = pdepe(m,@pdex1pde,@pdex1ic,@pdex1bc,x,t);
% Extract the first solution component as u.
```

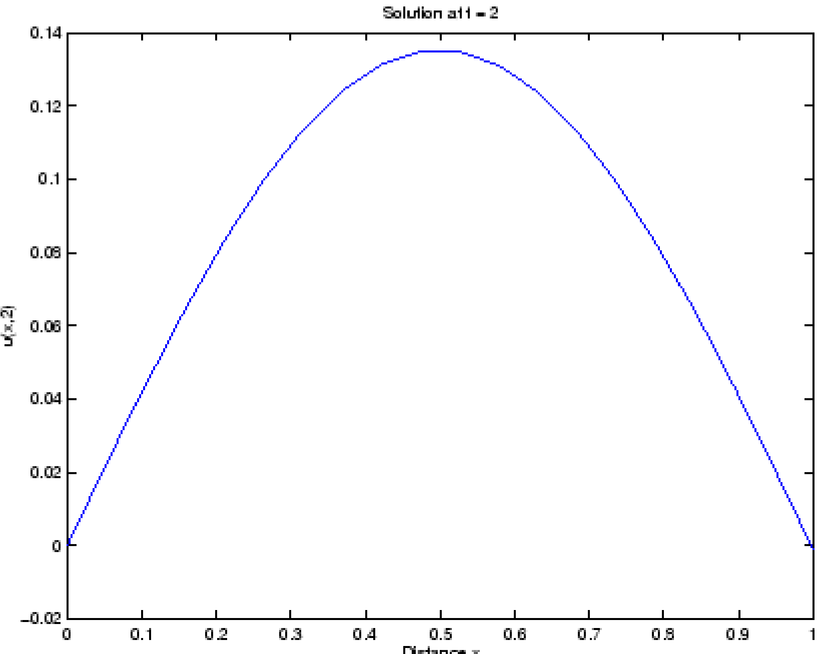
```
u = sol(:,:,1);

% A surface plot is often a good way to study a solution.
surf(x,t,u)
title('Numerical solution computed with 20 mesh points.')
xlabel('Distance x')
ylabel('Time t')

% A solution profile can also be illuminating.
figure
plot(x,u(end,:))
title('Solution at t = 2')
xlabel('Distance x')
ylabel('u(x,2)')
% --------------------------------------------------------------
function [c,f,s] = pdex1pde(x,t,u,DuDx)
c = pi^2;
f = DuDx;
s = 0;
% --------------------------------------------------------------
function u0 = pdex1ic(x)
u0 = sin(pi*x);
% --------------------------------------------------------------
function [pl,ql,pr,qr] = pdex1bc(xl,ul,xr,ur,t)
pl = ul;
ql = 0;
pr = pi * exp(-t);
qr = 1;
```

In this example, the PDE, initial condition, and boundary conditions are coded in local functions `pdex1pde`, `pdex1ic`, and `pdex1bc`.

The surface plot shows the behavior of the solution.



The following plot shows the solution profile at the final value of `t` (i.e., `t = 2`).



Example 2. This example illustrates the solution of a system of PDEs. The problem has boundary layers at both ends of the interval. The solution changes rapidly for small t.

The PDEs are

$$\frac{\partial u_1}{\partial t} = 0.024\frac{\partial^2 u_1}{\partial x^2} - F(u_1 - u_2)$$

$$\frac{\partial u_2}{\partial t} = 0.170\frac{\partial^2 u_2}{\partial x^2} + F(u_1 - u_2)$$

where $F(y) = \exp(5.73y) - \exp(-11.46y)$.

This equation holds on an interval $0 \le x \le 1$ for times $t \ge 0$.

The PDE satisfies the initial conditions

$$u_1(x,0) \equiv 1$$
$$u_2(x,0) \equiv 0$$

and boundary conditions

$$\frac{\partial u_1}{\partial x}(0,t) \equiv 0$$
$$u_2(0,t) \equiv 0$$
$$u_1(1,t) \equiv 1$$
$$\frac{\partial u_2}{\partial x}(1,t) \equiv 0$$

In the form expected by `pdepe`, the equations are

$$\begin{bmatrix}1\\1\end{bmatrix} .* \frac{\partial}{\partial t}\begin{bmatrix}u_1\\u_2\end{bmatrix} = \frac{\partial}{\partial x}\begin{bmatrix}0.024(\partial u_1/\partial x)\\0.170(\partial u_2/\partial x)\end{bmatrix} + \begin{bmatrix}-F(u_1-u_2)\\F(u_1-u_2)\end{bmatrix}$$

The boundary conditions on the partial derivatives of u have to be written in terms of the flux. In the form expected by `pdepe`, the left boundary condition is

$$\begin{bmatrix}0\\u_2\end{bmatrix} + \begin{bmatrix}1\\0\end{bmatrix} .* \begin{bmatrix}0.024(\partial u_1/\partial x)\\0.170(\partial u_2/\partial x)\end{bmatrix} = \begin{bmatrix}0\\0\end{bmatrix}$$

and the right boundary condition is

$$\begin{bmatrix}u_1-1\\0\end{bmatrix} + \begin{bmatrix}0\\1\end{bmatrix} .* \begin{bmatrix}0.024(\partial u_1/\partial x)\\0.170(\partial u_2/\partial x)\end{bmatrix} = \begin{bmatrix}0\\0\end{bmatrix}$$

The solution changes rapidly for small t. The program selects the step size in time to resolve this sharp change, but to see this behavior in the plots, the example must select the output times accordingly. There are boundary layers in the solution at both ends of [0,1], so the example places mesh points near `0` and `1` to resolve these sharp changes. Often

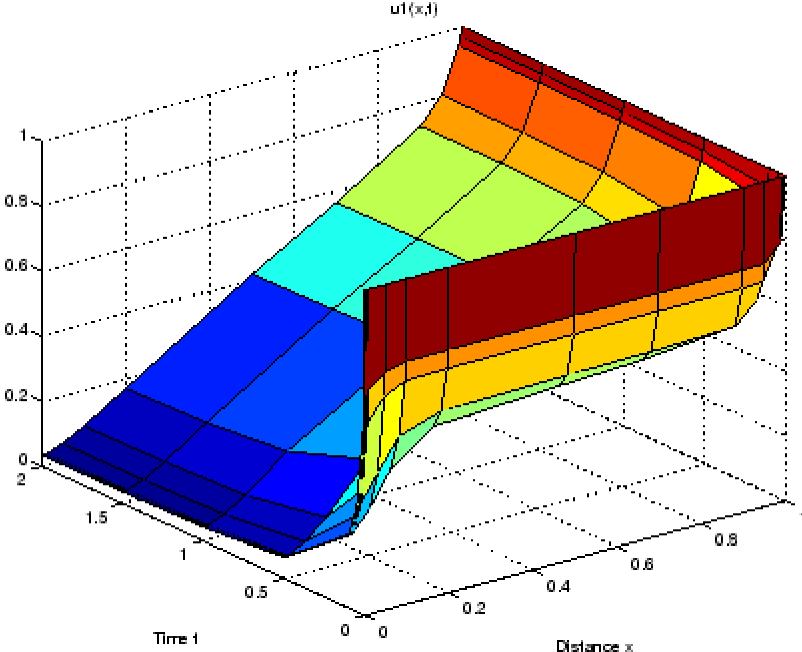
some experimentation is needed to select a mesh that reveals the behavior of the solution.

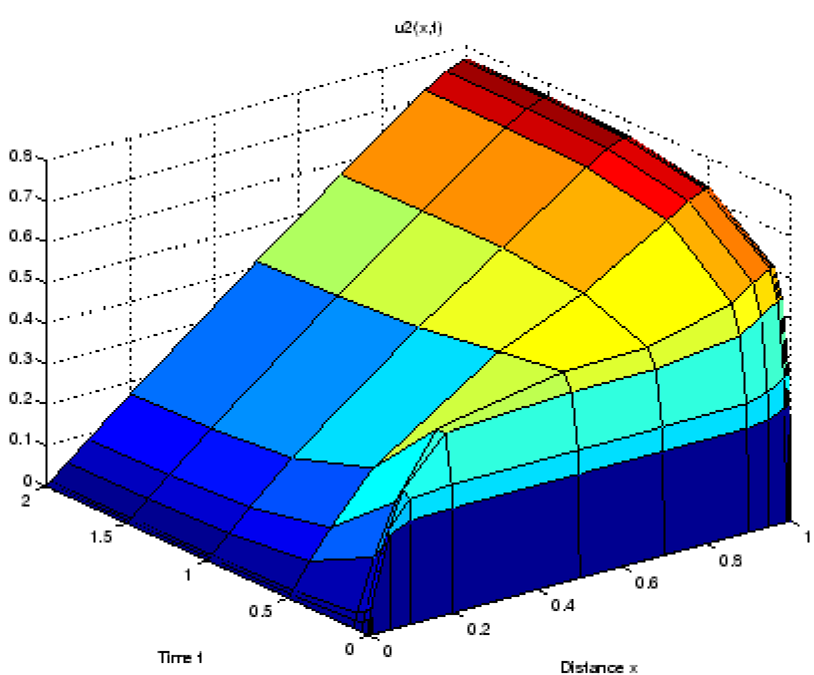
```
function pdex4
m = 0;
x = [0 0.005 0.01 0.05 0.1 0.2 0.5 0.7 0.9 0.95 0.99 0.995 1];
t = [0 0.005 0.01 0.05 0.1 0.5 1 1.5 2];

sol = pdepe(m,@pdex4pde,@pdex4ic,@pdex4bc,x,t);
u1 = sol(:,:,1);
u2 = sol(:,:,2);

figure
surf(x,t,u1)
title('u1(x,t)')
xlabel('Distance x')
ylabel('Time t')

figure
surf(x,t,u2)
title('u2(x,t)')
xlabel('Distance x')
ylabel('Time t')
% --------------------------------------------------------------
function [c,f,s] = pdex4pde(x,t,u,DuDx)
c = [1; 1];
f = [0.024; 0.17] .* DuDx;
y = u(1) - u(2);
F = exp(5.73*y)-exp(-11.47*y);
s = [-F; F];
% --------------------------------------------------------------
function u0 = pdex4ic(x);
u0 = [1; 0];
% --------------------------------------------------------------
function [pl,ql,pr,qr] = pdex4bc(xl,ul,xr,ur,t)
pl = [0; ul(2)];
ql = [1; 0];
```

```
pr = [ur(1)-1; 0];
qr = [0; 1];
```

In this example, the PDEs, initial conditions, and boundary conditions are coded in local functions `pdex4pde`, `pdex4ic`, and `pdex4bc`.

The surface plots show the behavior of the solution components.





References

[1] Skeel, R. D. and M. Berzins, "A Method for the Spatial Discretization of Parabolic Equations in One Space Variable," *SIAM Journal on Scientific and Statistical Computing*, Vol. 11, 1990, pp.1–32.

See Also

`function_handle` | `pdeval` | `ode15s` | `odeset` | `odeget`

Purpose

Evaluate numerical solution of PDE using output of pdepe

Syntax

```
[uout,duoutdx] = pdeval(m,x,ui,xout)
```

Arguments

`m`	Symmetry of the problem: slab = `0`, cylindrical = `1`, spherical = `2`. This is the first input argument used in the call to `pdepe`.
`xmesh`	A vector [`x0`, `x1`, ..., `xn`] specifying the points at which the elements of `ui` were computed. This is the same vector with which `pdepe` was called.
`ui`	A vector `sol(j,:,i)` that approximates component `i` of the solution at time t_f and mesh points `xmesh`, where `sol` is the solution returned by `pdepe`.
`xout`	A vector of points from the interval [`x0`,`xn`] at which the interpolated solution is requested.

Description

`[uout,duoutdx] = pdeval(m,x,ui,xout)` approximates the solution u_i and its partial derivative $\partial u_i/\partial x$ at points from the interval [`x0`,`xn`]. The `pdeval` function returns the computed values in `uout` and `duoutdx`, respectively.

Note `pdeval` evaluates the partial derivative $\partial u_i/\partial x$ rather than the flux f. Although the flux is continuous, the partial derivative may have a jump at a material interface.

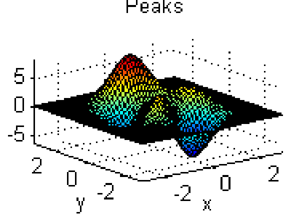
See Also

`pdepe`

peaks

Purpose

Example function of two variables



Syntax

```
Z = peaks;
Z = peaks(n);
Z = peaks(V);
Z = peaks(X,Y);

peaks;
peaks(N);
peaks(V);
peaks(X,Y);

[X,Y,Z] = peaks;
[X,Y,Z] = peaks(n);
[X,Y,Z] = peaks(V);
```

Description

`peaks` is a function of two variables, obtained by translating and scaling Gaussian distributions, which is useful for demonstrating `mesh`, `surf`, `pcolor`, `contour`, and so on.

`Z = peaks;` returns a 49-by-49 matrix.

`Z = peaks(n);` returns an `n`-by-`n` matrix.

`Z = peaks(V);` returns an `n`-by-`n` matrix, where `n = length(V)`.

`Z = peaks(X,Y);` evaluates `peaks` at the given `X` and `Y` (which must be the same size) and returns a matrix the same size.

`peaks(...)` (with no output argument) plots the peaks function with `surf`.

`[X,Y,Z] = peaks(...);` returns two additional matrices, `X` and `Y`, for parametric plots, for example, `surf(X,Y,Z,del2(Z))`. If not given as input, the underlying matrices `X` and `Y` are

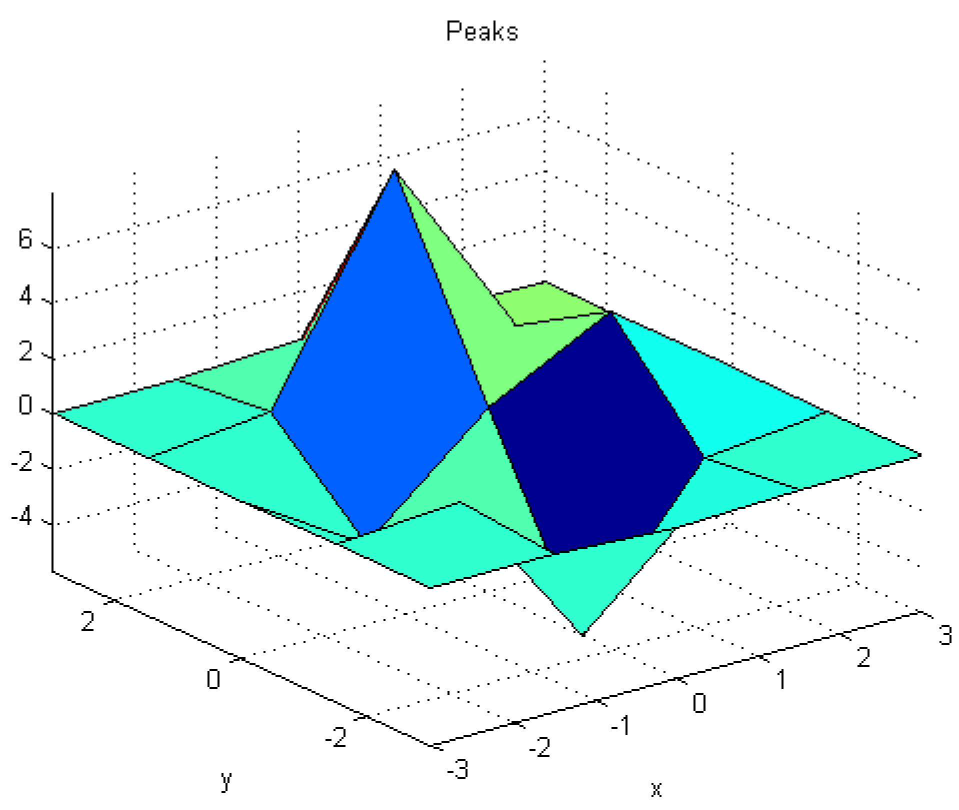
```
[X,Y] = meshgrid(V,V)
```

where `V` is a given vector, or `V` is a vector of length `n` with elements equally spaced from -3 to 3. If no input argument is given, the default `n` is 49.

Examples

This example creates a 5–by–5 matrix of peaks and displays the image:

```
peaks(5);
```



See Examples section from `surf` for a more elaborate use case.

See Also

`meshgrid` | `surf`

Purpose

Call Perl script using appropriate operating system executable

Syntax

```
perl('perlfile')
perl('perlfile',arg1,arg2,...)
result = perl(...)
[result, status] = perl(...)
```

Description

`perl('perlfile')` calls the Perl script `perlfile`, using the appropriate operating system Perl executable. Perl is included with the MATLAB software on Microsoft Windows systems, and thus MATLAB users can run user-created MATLAB functions containing the `perl` function. On Linux and Macintosh systems, MATLAB calls the Perl interpreter available with the operating system.

`perl('perlfile',arg1,arg2,...)` calls the Perl script `perlfile`, using the appropriate operating system Perl executable, and passes the arguments `arg1`, `arg2`, and so on, to `perlfile`.

`result = perl(...)` returns the results of attempted Perl call to `result`.

`[result, status] = perl(...)` returns the results of attempted Perl call to `result` and its exit status to `status`.

It is sometimes beneficial to use Perl scripts instead of MATLAB code. The `perl` function allows you to run those scripts from MATLAB. Specific examples where you might choose to use a Perl script include:

- Perl script already exists
- Perl script preprocesses data quickly, formatting it in a way more easily read by MATLAB
- Perl has features not supported by MATLAB

Examples

Given the Perl script, `hello.pl`:

```
$input = $ARGV[0];
print "Hello $input.";
```

At the MATLAB command line, type:

```
perl('hello.pl','World')
```

MATLAB displays:

```
ans =
Hello World.
```

See Also

dos | regexp | system | unix | ! (exclamation point)

Purpose

All possible permutations

Syntax

```
P = perms(v)
```

Description

`P = perms(v)` returns a matrix containing all permutations of the elements of vector `v` in reverse lexicographic order. Each row of `P` contains a different permutation of the *n* elements in `v`. Matrix `P` has the same data type as `v`, and it has *n*! rows and *n* columns.

Input Arguments

v - Set of items

vector of numeric, `logical`, or `char` values

Set of items, specified as a vector of numeric, `logical`, or `char` values.

Example: `[1 2 3 4]`

Example: `[1+1i 2+1i 3+1i 4+1i]`

Example: `int16([1 2 3 4])`

Example: `['abcd']`

Example: `[true false true false]`

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

Limitations

`perms(v)` is practical when `length(v)` is less than about 10.

Examples

All Permutations of Double Integers

```
v = [2 4 6];
P = perms(v)
```

```
P =

     6     4     2
     6     2     4
```

```
     4     6     2
     4     2     6
     2     4     6
     2     6     4
```

All Permutations of Unsigned Integers

```
v = uint16([1023 4095 65535]);
P = perms(v)

P =

  65535   4095   1023
  65535   1023   4095
   4095  65535   1023
   4095   1023  65535
   1023   4095  65535
   1023  65535   4095
```

All Permutations of Complex Numbers

```
v = [1+1i 2+1i 3+1i];
P = perms(v)

P =

   3.0000 + 1.0000i   2.0000 + 1.0000i   1.0000 + 1.0000i
   3.0000 + 1.0000i   1.0000 + 1.0000i   2.0000 + 1.0000i
   2.0000 + 1.0000i   3.0000 + 1.0000i   1.0000 + 1.0000i
   2.0000 + 1.0000i   1.0000 + 1.0000i   3.0000 + 1.0000i
   1.0000 + 1.0000i   2.0000 + 1.0000i   3.0000 + 1.0000i
   1.0000 + 1.0000i   3.0000 + 1.0000i   2.0000 + 1.0000i
```

See Also

nchoosek | permute | randperm

Purpose

Rearrange dimensions of N-D array

Syntax

```
B = permute(A,order)
```

Description

`B = permute(A,order)` rearranges the dimensions of `A` so that they are in the order specified by the vector `order`. `B` has the same values of `A` but the order of the subscripts needed to access any particular element is rearranged as specified by `order`. All the elements of `order` must be unique, real, positive, integer values.

Tips

`permute` and `ipermute` are a generalization of transpose (`.'`) for multidimensional arrays.

Examples

Given any matrix `A`, the statement

```
permute(A,[2 1])
```

is the same as `A.'`.

For example:

```
A = [1 2; 3 4]; permute(A,[2 1])
ans =
     1     3
     2     4
```

The following code permutes a three-dimensional array:

```
X = rand(12,13,14);
Y = permute(X,[2 3 1]);
size(Y)
ans =
    13    14    12
```

See Also

`ipermute` | `circshift` | `fliplr` | `flipud` | `reshape` | `shiftdim`

persistent

Purpose

Define persistent variable

Syntax

```
persistent X Y Z
```

Description

`persistent X Y Z` defines `X`, `Y`, and `Z` as variables that are local to the function in which they are declared; yet their values are retained in memory between calls to the function. Persistent variables are similar to global variables because the MATLAB software creates permanent storage for both. They differ from global variables in that persistent variables are known only to the function in which they are declared. This prevents persistent variables from being changed by other functions or from the MATLAB command line.

Whenever you clear or modify a function that is in memory, MATLAB also clears all persistent variables declared by that function. To keep a function in memory until MATLAB quits, use `mlock`.

If the persistent variable does not exist the first time you issue the `persistent` statement, it is initialized to the empty matrix.

It is an error to declare a variable persistent if a variable with the same name exists in the current workspace. MATLAB also errors if you declare any of a function's input or output arguments as persistent within that same function. For example, the following persistent declaration is invalid:

```
function myfun(argA, argB, argC)
persistent argB
```

Tips

There is no function form of the `persistent` command (i.e., you cannot use parentheses and quote the variable names).

Examples

This function writes a large array to a spreadsheet file and then reads several rows from the same file. Because you only need to write the array to the spreadsheet one time, the program tests whether an array can be read from the file and, if so, does not waste time in repeating that task. By defining the `dblArray` variable as persistent, you can easily check whether the array has been read from the spreadsheet file.

Here is the `arrayToXLS` function:

```
function arrayToXLS(A, xlsfile, x1, x2)
persistent dblArray;

if isempty(dblArray)
    disp 'Writing spreadsheet file ...'
    xlswrite(xlsfile, A);
end

disp 'Reading array from spreadsheet ...'
dblArray = xlsread(xlsfile, 'Sheet1', [x1 ':' x2])
fprintf('\n');
```

Run the function three times and observe the time elapsed for each run. The second and third run take approximately one tenth the time of the first run in which the function must create the spreadsheet:

```
largeArray = rand(4000, 200);

tic,  arrayToXLS(largeArray, 'myTest.xls','E254', 'J256'),  toc
Writing spreadsheet file ...
Reading array from spreadsheet ...
dblArray =
    0.0982    0.3783    0.1264    0.7880    0.1902    0.5811
    0.2251    0.2704    0.5682    0.7271    0.8028    0.2834
    0.6453    0.5568    0.8254    0.4961    0.9096    0.5402

Elapsed time is 8.990525 seconds.


tic,  arrayToXLS(largeArray, 'myTest.xls','E257', 'J258'),  toc
Reading array from spreadsheet ...
dblArray =
    0.4620    0.3781    0.6386    0.5930    0.0946    0.4865
    0.1605    0.1251    0.8709    0.5188    0.6702    0.2138
```

```
Elapsed time is 0.912534 seconds.


tic,  arrayToXLS(largeArray, 'myTest.xls','E259', 'J262'),  toc
Reading array from spreadsheet ...
dblArray =
    0.7015    0.6588    0.4023    0.0359    0.4512    0.6097
    0.1308    0.6441    0.0431    0.6396    0.7481    0.8688
    0.8278    0.2686    0.5475    0.8550    0.5896    0.1080
    0.9437    0.1671    0.0505    0.1203    0.2461    0.7306

Elapsed time is 0.928843 seconds.
```

Now clear the `arrayToXLS` function from memory and observe that running it takes much longer again:

```
clear functions

tic,  arrayToXLS(largeArray, 'myTest.xls','E263', 'J264'),  toc
Writing spreadsheet file ...
Reading array from spreadsheet ...
dblArray =
    0.6292    0.7788    0.0732    0.6481    0.9299    0.8631
    0.7700    0.5181    0.9805    0.5092    0.8658    0.4070

Elapsed time is 7.603461 seconds.
```

See Also

`global` | `clear` | `mislocked` | `mlock` | `munlock` | `isempty`

Purpose

Ratio of circle's circumference to its diameter

Syntax

```
pi
```

Description

`pi` returns the floating-point number nearest the value of π. The expressions `4*atan(1)` and `imag(log(-1))` provide the same value.

Examples

Find the sine of π:

```
sin(pi)
```

returns

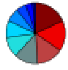
```
ans =

   1.2246e-16
```

The expression `sin(pi)` is not exactly zero because `pi` is not exactly π.

pie

Purpose

Pie chart



Syntax

```
pie(X)
pie(X,explode)
pie(...,labels)
pie(axes_handle,...)
h = pie(...)
```

Description

`pie(X)` draws a pie chart using the data in `X`. Each element in `X` is represented as a slice in the pie chart.

`pie(X,explode)` offsets a slice from the pie. `explode` is a vector or matrix of zeros and nonzeros that correspond to `X`. A nonzero value offsets the corresponding slice from the center of the pie chart, so that `X(i,j)` is offset from the center if `explode(i,j)` is nonzero. `explode` must be the same size as `X`.

`pie(...,labels)` specifies text labels for the slices. The number of labels must equal the number of elements in `X`. For example,

```
pie(1:3,{'Taxes','Expenses','Profit'})
```

`pie(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = pie(...)` returns a vector of handles to patch and text graphics objects.

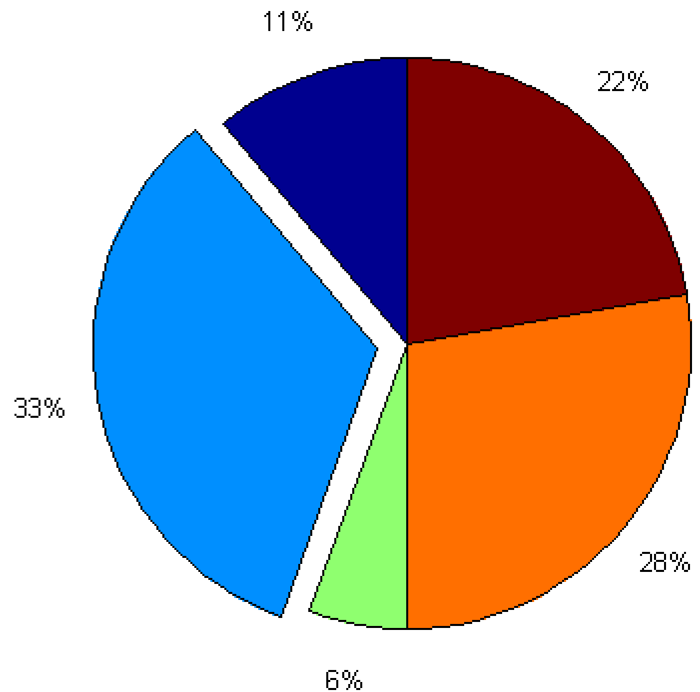
Tips

The values in `X` are normalized via `X/sum(X)` to determine the area of each slice of the pie. If `sum(X)`   `1`, the values in `X` directly specify the area of the pie slices. MATLAB draws only a partial pie if `sum(X) < 1`.

Examples

Emphasize the second slice in the chart by setting its corresponding `explode` element to 1.

```
x = [1 3 0.5 2.5 2];
explode = [0 1 0 0 0];
pie(x,explode)
colormap jet
```



See Also

`pie3`

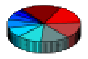
How To

- “Pie Charts”

pie3

Purpose

3-D pie chart



Syntax

```
pie3(X)
pie3(X,explode)
pie3(...,labels)
pie3(axes_handle,...)
h = pie3(...)
```

Description

`pie3(X)` draws a three-dimensional pie chart using the data in `X`. Each element in `X` is represented as a slice in the pie chart.

`pie3(X,explode)` specifies whether to offset a slice from the center of the pie chart. `X(i,j)` is offset from the center of the pie chart if `explode(i,j)` is nonzero. `explode` must be the same size as `X`.

`pie3(...,labels)` specifies text labels for the slices. The number of labels must equal the number of elements in `X`. For example,

```
pie3(1:3,{'Taxes','Expenses','Profit'})
```

`pie3(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = pie3(...)` returns a vector of handles to patch, surface, and text graphics objects.

Tips

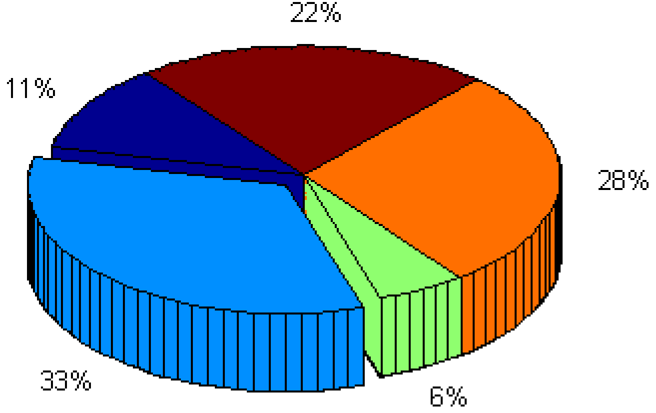
The values in `X` are normalized via `X/sum(X)` to determine the area of each slice of the pie. If `sum(X)`   `1`, the values in `X` directly specify the area of the pie slices. MATLAB draws only a partial pie if `sum(X) < 1`.

Examples

Offset a slice in the pie chart by setting the corresponding `explode` element to `1`:

```
x = [1 3 0.5 2.5 2];
explode = [0 1 0 0 0];
```

```
pie3(x,explode)
colormap hsv
```



See Also

`pie`

Purpose

Moore-Penrose pseudoinverse of matrix

Syntax

```
B = pinv(A)
B = pinv(A,tol)
```

Definitions

The Moore-Penrose pseudoinverse is a matrix `B` of the same dimensions as `A'` satisfying four conditions:

```
A*B*A = A
B*A*B = B
A*B is Hermitian
B*A is Hermitian
```

The computation is based on `svd(A)` and any singular values less than `tol` are treated as zero.

Description

`B = pinv(A)` returns the Moore-Penrose pseudoinverse of `A`.

`B = pinv(A,tol)` returns the Moore-Penrose pseudoinverse and overrides the default tolerance, `max(size(A))*norm(A)*eps`.

Examples

If `A` is square and not singular, then `pinv(A)` is an expensive way to compute `inv(A)`. If `A` is not square, or is square and singular, then `inv(A)` does not exist. In these cases, `pinv(A)` has some of, but not all, the properties of `inv(A)`.

If `A` has more rows than columns and is not of full rank, then the overdetermined least squares problem

```
minimize norm(A*x-b)
```

does not have a unique solution. Two of the infinitely many solutions are

```
x = pinv(A)*b
```

and

```
y = A\b
```

These two are distinguished by the facts that `norm(x)` is smaller than the norm of any other solution and that `y` has the fewest possible nonzero components.

For example, the matrix generated by

```
A = magic(8); A = A(:,1:6)
```

is an 8-by-6 matrix that happens to have `rank(A) = 3`.

```
A =
    64     2     3    61    60     6
     9    55    54    12    13    51
    17    47    46    20    21    43
    40    26    27    37    36    30
    32    34    35    29    28    38
    41    23    22    44    45    19
    49    15    14    52    53    11
     8    58    59     5     4    62
```

The right-hand side is `b = 260*ones(8,1)`,

```
b =
     260
     260
     260
     260
     260
     260
     260
     260
```

The scale factor 260 is the 8-by-8 magic sum. With all eight columns, one solution to `A*x = b` would be a vector of all `1`'s. With only six columns, the equations are still consistent, so a solution exists, but it is not all `1`'s. Since the matrix is rank deficient, there are infinitely many solutions. Two of them are

```
x = pinv(A)*b
```

which is

```
x =
    1.1538
    1.4615
    1.3846
    1.3846
    1.4615
    1.1538
```

and

```
y = A\b
```

which produces this result.

```
Warning: Rank deficient, rank = 3  tol =   1.8829e-013.
y =
    4.0000
    5.0000
         0
         0
         0
   -1.0000
```

Both of these are exact solutions in the sense that `norm(A*x-b)` and `norm(A*y-b)` are on the order of roundoff error. The solution `x` is special because

```
norm(x) = 3.2817
```

is smaller than the norm of any other solution, including

```
norm(y) = 6.4807
```

On the other hand, the solution `y` is special because it has only three nonzero components.

See Also

`inv` | `qr` | `rank` | `svd`

Purpose

Givens plane rotation

Syntax

```
[G,y] = planerot(x)
```

Description

`[G,y] = planerot(x)` where `x` is a 2-component column vector, returns a 2-by-2 orthogonal matrix G so that `y = G*x` has `y(2) = 0`.

Examples

```
x = [3 4];
[G,y] = planerot(x')

G  =
    0.6000    0.8000
   -0.8000    0.6000

y =
     5
     0
```

See Also

`qrdelete` | `qrinsert`

audioplayer.play

Purpose

Play audio from `audioplayer` object

Syntax

```
play(playerObj)
play(playerObj,start)
play(playerObj,[start,stop])
```

Description

`play(playerObj)` plays the audio associated with `audioplayer` object `playerObj` from beginning to end.

`play(playerObj,start)` plays audio from the sample indicated by `start` to the end.

`play(playerObj,[start,stop])` plays audio from the sample indicated by `start` to the sample indicated by `stop`.

Examples

Play with and without Blocking

Play two audio samples with and without blocking using the `play` and `playblocking` methods.

Load data from example files `chirp.mat` and `gong.mat`.

```
chirpData = load('chirp.mat');
chirpObj = audioplayer(chirpData.y,chirpData.Fs);

gongData = load('gong.mat');
gongObj = audioplayer(gongData.y,gongData.Fs);
```

Play the samples with blocking, one after the other.

```
playblocking(chirpObj);
playblocking(gongObj);
```

Play without blocking. The audio can overlap.

```
play(chirpObj);
play(gongObj);
```

Starting Sample

Play audio from the example file `handel.mat` starting 4 seconds from the beginning.

```
load handel.mat;
playerObj = audioplayer(y,Fs);
start = playerObj.SampleRate * 4;

play(playerObj,start);
```

Sample Range

Play the first 3 seconds of audio from the example file `handel.mat`.

```
load handel.mat;
playerObj = audioplayer(y,Fs);
start = 1;
stop = playerObj.SampleRate * 3;

play(playerObj,[start,stop]);
```

See Also

`audioplayer` | `playblocking`

How To

- “Play Audio”

audiorecorder.play

Purpose

Play audio from `audiorecorder` object

Syntax

```
player = play(recObj)
player = play(recObj, start)
player = play(recObj, [start stop])
```

Description

`player` = `play(`*`recObj`*`)` plays the audio associated with `audiorecorder` object *`recObj`* from beginning to end, and returns an `audioplayer` object.

`player` = `play(`*`recObj`*`, `*`start`*`)` plays audio from the sample indicated by *`start`* to the end.

`player` = `play(`*`recObj`*`, [`*`start stop`*`])` plays audio from the sample indicated by *`start`* to the sample indicated by *`stop`*.

Examples

Record 5 seconds of your speech with a microphone, and play it back. Display the properties of the `audioplayer` object.

```
myVoice = audiorecorder;

disp('Start speaking.');
recordblocking(myVoice, 5);
disp('End of recording. Playing back ...');

playerObj = play(myVoice);

disp('Properties of playerObj:');
get(playerObj)
```

Play back only the first 3 seconds of the speech recorded in the previous example:

```
play(myVoice, [1 myVoice.SampleRate*3]);
```

See Also

`audioplayer` | `audiorecorder`

Purpose

Play audio from `audioplayer` object, holding control until playback completes

Syntax

```
playblocking(playerObj)
playblocking(playerObj,start)
playblocking(playerObj,[start,stop])
```

Description

`playblocking(playerObj)` plays the audio associated with `audioplayer` object `playerObj` from beginning to end. `playblocking` does not return control until playback completes.

`playblocking(playerObj,start)` plays audio from the sample indicated by `start` to the end.

`playblocking(playerObj,[start,stop])` plays audio from the sample indicated by `start` to the sample indicated by `stop`.

Examples

Play with and without Blocking

Play two audio samples with and without blocking using the `play` and `playblocking` methods.

Load data from example files `chirp.mat` and `gong.mat`.

```
chirpData = load('chirp.mat');
chirpObj = audioplayer(chirpData.y,chirpData.Fs);

gongData = load('gong.mat');
gongObj = audioplayer(gongData.y,gongData.Fs);
```

Play the samples with blocking, one after the other.

```
playblocking(chirpObj);
playblocking(gongObj);
```

Play without blocking. The audio can overlap.

```
play(chirpObj);
play(gongObj);
```

Starting Sample

Play audio from the example file `handel.mat` starting 4 seconds from the beginning.

```
load handel.mat;
playerObj = audioplayer(y,Fs);
start = playerObj.SampleRate * 4;

playblocking(playerObj,start);
beep;
```

Sample Range

Play the first 3 seconds of audio from the example file `handel.mat`.

```
load handel.mat;
playerObj = audioplayer(y,Fs);
start = 1;
stop = playerObj.SampleRate * 3;

playblocking(playerObj,[start,stop]);
beep;
```

See Also

`audioplayer` | `play`

How To

- "Play Audio"

Purpose

2-D line plot

Syntax

```
plot(Y)
plot(X1,Y1,...,Xn,Yn)
plot(X1,Y1,LineSpec,...,Xn,Yn,LineSpec)
plot(...,'PropertyName',PropertyValue,...)
plot(axes_handle,...)
h = plot(...)
```

Description

`plot(Y)` plots the columns of `Y` versus the index of each value when `Y` is a real number. For complex `Y`, `plot(Y)` is equivalent to `plot(real(Y),imag(Y))`.

`plot(X1,Y1,...,Xn,Yn)` plots each vector `Yn` versus vector `Xn` on the same axes. If one of `Yn` or `Xn` is a matrix and the other is a vector, it plots the vector versus the matrix row or column with a matching dimension to the vector. If `Xn` is a scalar and `Yn` is a vector, it plots discrete `Yn` points vertically at `Xn`. If `Xn` or `Yn` are complex, imaginary components are ignored. If `Xn` or `Yn` are matrices, they must be 2-D and the same size, and the columns of `Yn` are plotted against the columns of `Xn`. `plot` automatically chooses colors and line styles in the order specified by `ColorOrder` and `LineStyleOrder` properties of current axes.

`plot(X1,Y1,LineSpec,...,Xn,Yn,LineSpec)` plots lines defined by the `Xn,Yn,LineSpec` triplets, where `LineSpec` specifies the line type, marker symbol, and color. You can mix `Xn,Yn,LineSpec` triplets with `Xn,Yn` pairs: `plot(X1,Y1,X2,Y2,LineSpec,X3,Y3)`.

`plot(...,'PropertyName',PropertyValue,...)` manipulates plot characteristics by setting `lineseries properties` (of `lineseries` graphics objects created by `plot`). Enter properties as one or more name-value pairs. Property name-value pairs apply to all the lines plotted. You cannot specify name-value pairs for each set of data.

`plot(axes_handle,...)` plots using axes with the handle `axes_handle` instead of the current axes (`gca`).

`h = plot(...)` returns a column vector of handles to `lineseries` objects, one handle per line.

Tips

Plot data can include `NaN` and `inf` values, which cause breaks in the lines drawn. For example,

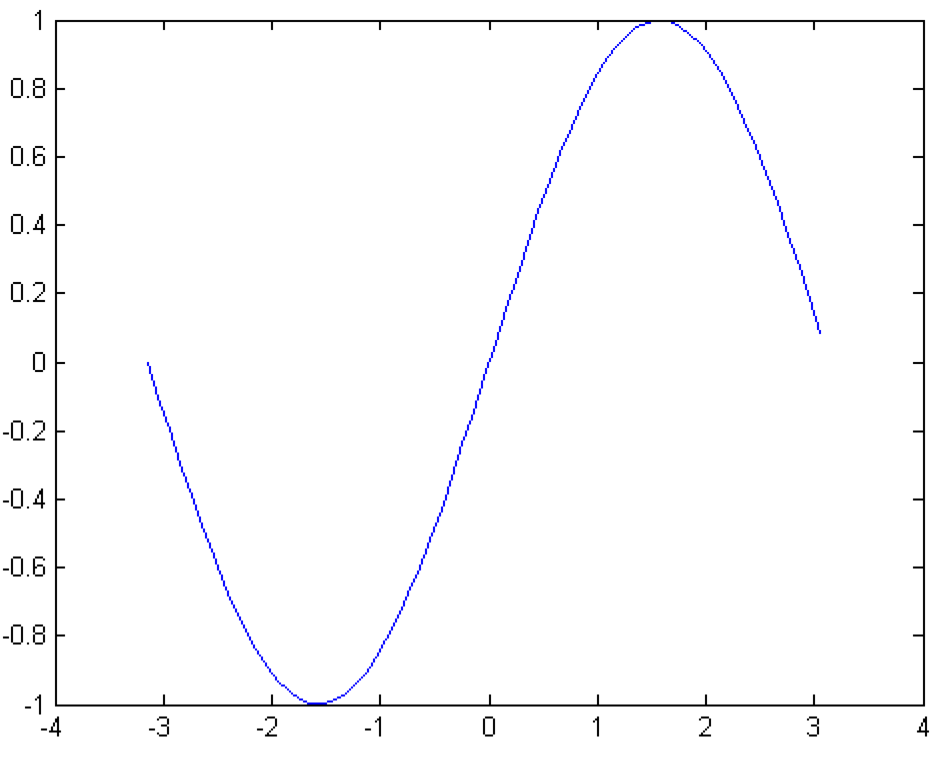
```
plot([1:5,NaN,6:10])
```

Skips the sixth element and resumes line drawing at the seventh element with the `Y` value of 6.

Examples

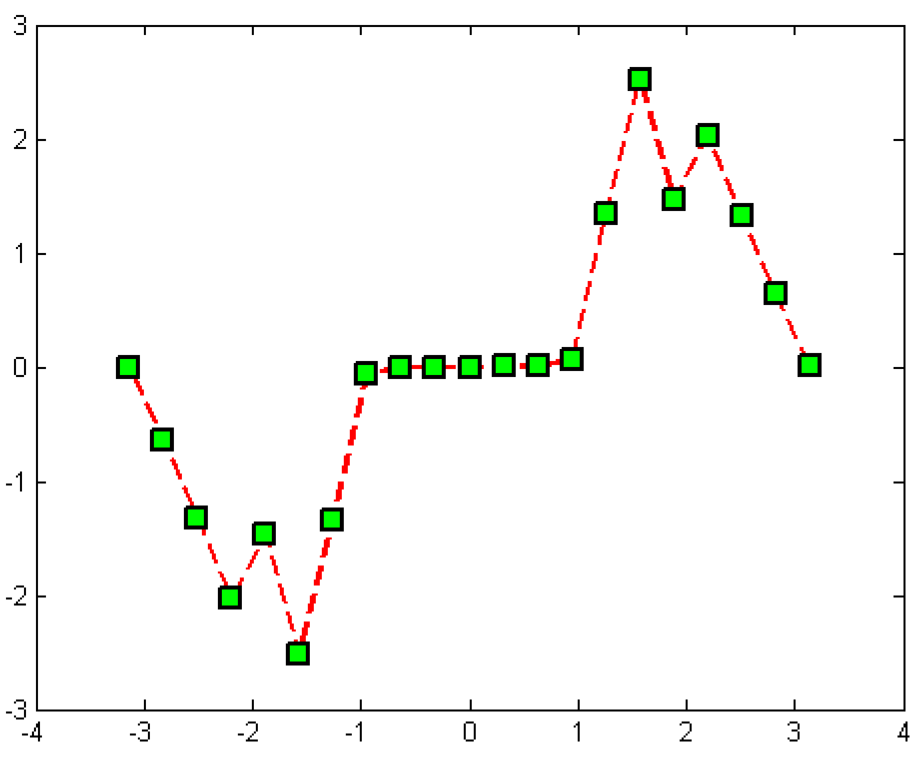
Plot a sine curve.

```
x = -pi:.1:pi;
y = sin(x);
plot(x,y)
```



Create line plot using specific line width, marker color, and marker size.

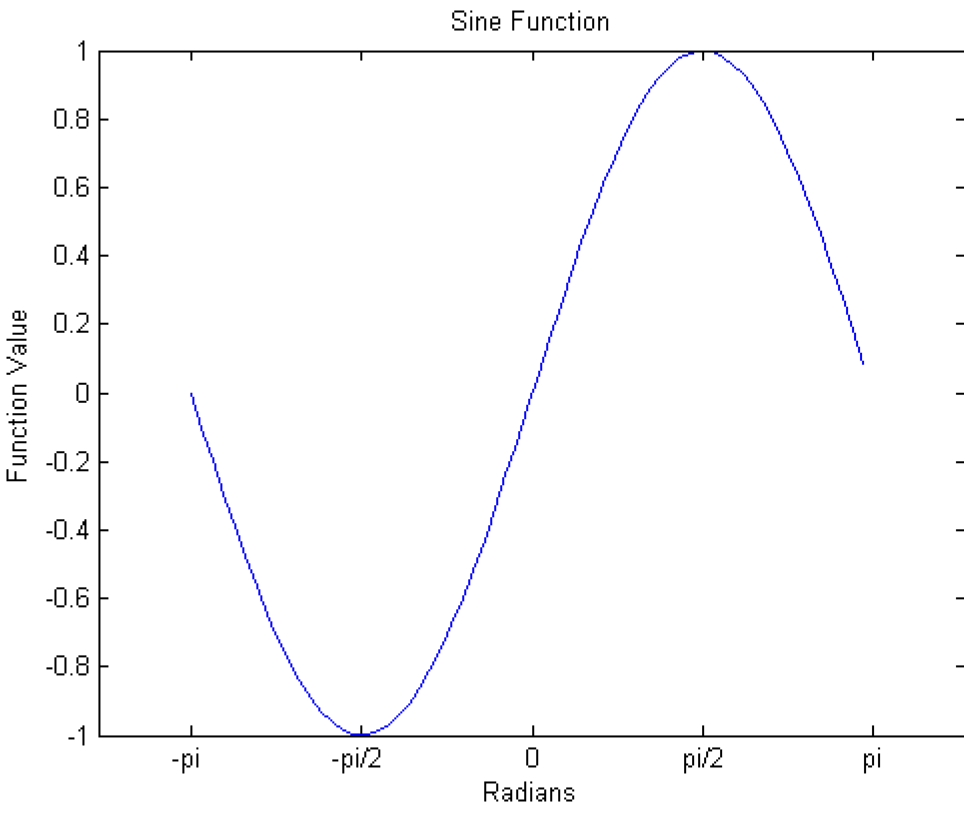
```
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));
plot(x,y,'--rs','LineWidth',2,...
                'MarkerEdgeColor','k',...
                'MarkerFaceColor','g',...
                'MarkerSize',10)
```



Modify axis tick marks and tick labels and annotate the graph.

```
x = -pi:.1:pi;
y = sin(x);
plot(x,y)
set(gca,'XTick',-pi:pi/2:pi)
set(gca,'XTickLabel',{'-pi','-pi/2','0','pi/2','pi'})
```

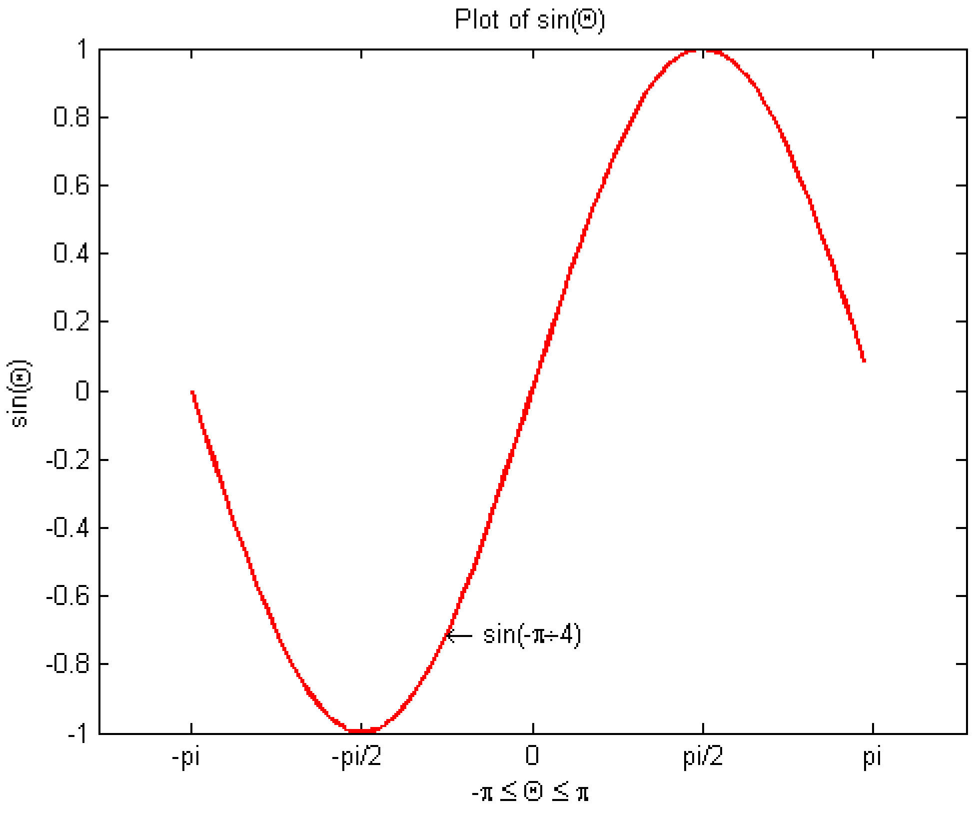
```
title('Sine Function');
xlabel('Radians');
ylabel('Function Value');
```



Add a plot title, axis labels, and annotations.

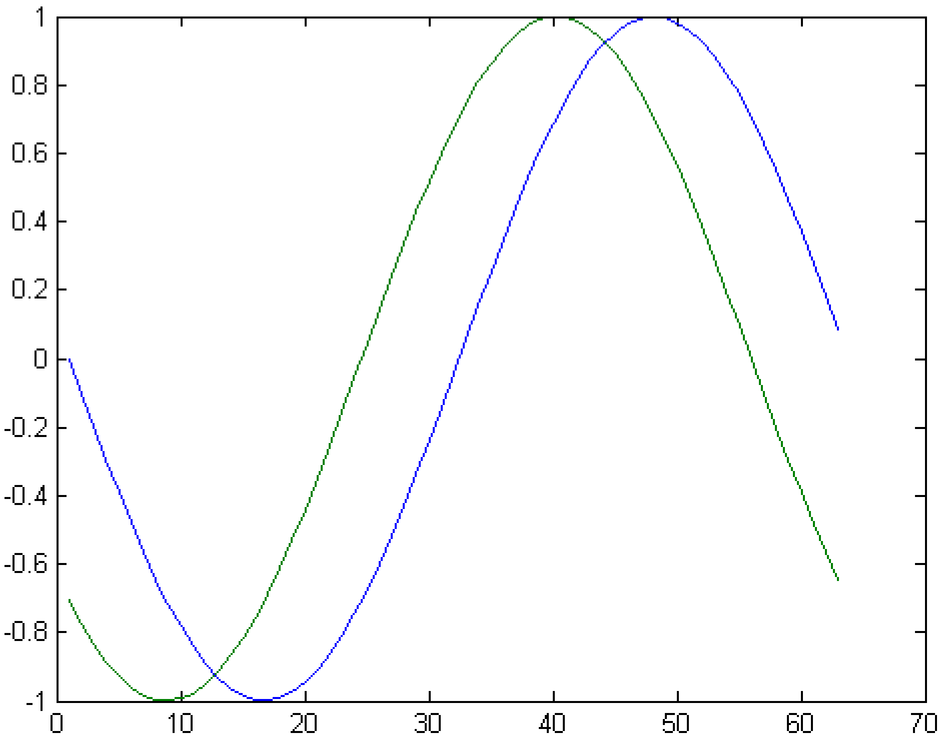
```
x = -pi:.1:pi;
y = sin(x);
p = plot(x,y)
```

```
set(gca,'XTick',-pi:pi/2:pi)
set(gca,'XTickLabel',{'-pi','-pi/2','0','pi/2','pi'})
xlabel('-\pi \leq \Theta \leq \pi')
ylabel('sin(\Theta)')
title('Plot of sin(\Theta)')
% \Theta appears as a Greek symbol (see String)
% Annotate the point (-pi/4, sin(-pi/4))
text(-pi/4,sin(-pi/4),'\leftarrow sin(-\pi\div4)',...
     'HorizontalAlignment','left')
% Change the line color to red and
% set the line width to 2 points
set(p,'Color','red','LineWidth',2)
```



Plot multiple line plots on the same axes.

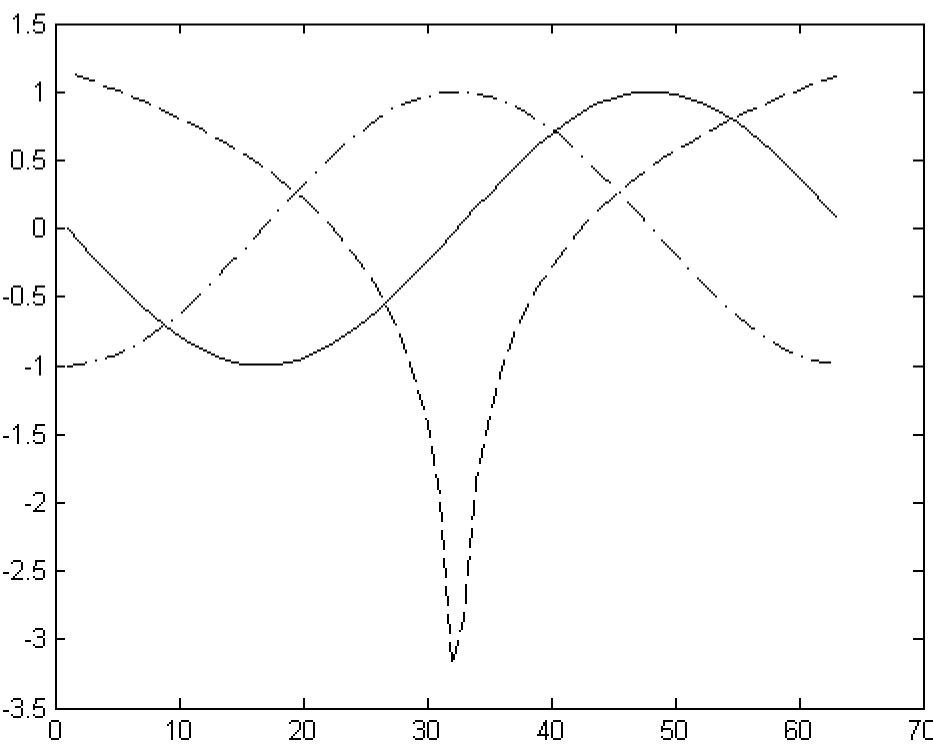
```
plot(sin(x));
% hold axes and all lineseries properties, such as
% ColorOrder and LineStyleOrder, for the next plot
hold all
plot(sin(x+(pi/4)));
```



Set line color to be always black and line style order to cycle through solid, dash-dot, dash-dash, and dotted line styles.

```
set(0,'DefaultAxesColorOrder',[0 0 0],...
      'DefaultAxesLineStyleOrder','-|-.|--|:')
plot(sin(x))
hold all
plot(cos(x))
hold all
```

```
plot(log(abs(x)))
```



See Also

axis | axes | bar | gca | grid | hold | legend | line | lineseries properties | LineSpec | LineWidth | loglog | MarkerEdgeColor | MarkerFaceColor | MarkerSize | plot3 | plotyy | semilogx | semilogy | subplot | title | xlabel | xlim | ylabel | ylim

How To

- Editing Plot Characteristics
- Creating Line Plots

- Annotating Graphs
- “Axes Objects — Defining Coordinate Systems for Graphs”

Purpose

3-D line plot

Syntax

```
plot3(X1,Y1,Z1,...)
plot3(X1,Y1,Z1,LineSpec,...)
plot3(...,'PropertyName',PropertyValue,...)
h = plot3(...)
```

Description

The `plot3` function displays a three-dimensional plot of a set of data points.

`plot3(X1,Y1,Z1,...)`, where `X1`, `Y1`, `Z1` are vectors or matrices, plots one or more lines in three-dimensional space through the points whose coordinates are the elements of `X1`, `Y1`, and `Z1`.

`plot3(X1,Y1,Z1,LineSpec,...)` creates and displays all lines defined by the `Xn,Yn,Zn,LineSpec` quads, where `LineSpec` is a line specification that determines line style, marker symbol, and color of the plotted lines.

`plot3(...,'`*`PropertyName`*`',PropertyValue,...)` sets properties to the specified property values for all line graphics objects created by `plot3`. See `lineseries properties` for a description of the properties you can set.

`h = plot3(...)` returns a column vector of handles to lineseries graphics objects, with one handle per object.

Tips

If one or more of `X1`, `Y1`, `Z1` is a vector, the vectors are plotted versus the rows or columns of the matrix, depending whether the vectors' lengths equal the number of rows or the number of columns.

You can mix `Xn,Yn,Zn` triples with `Xn,Yn,Zn,`*`LineSpec`* quads, for example,

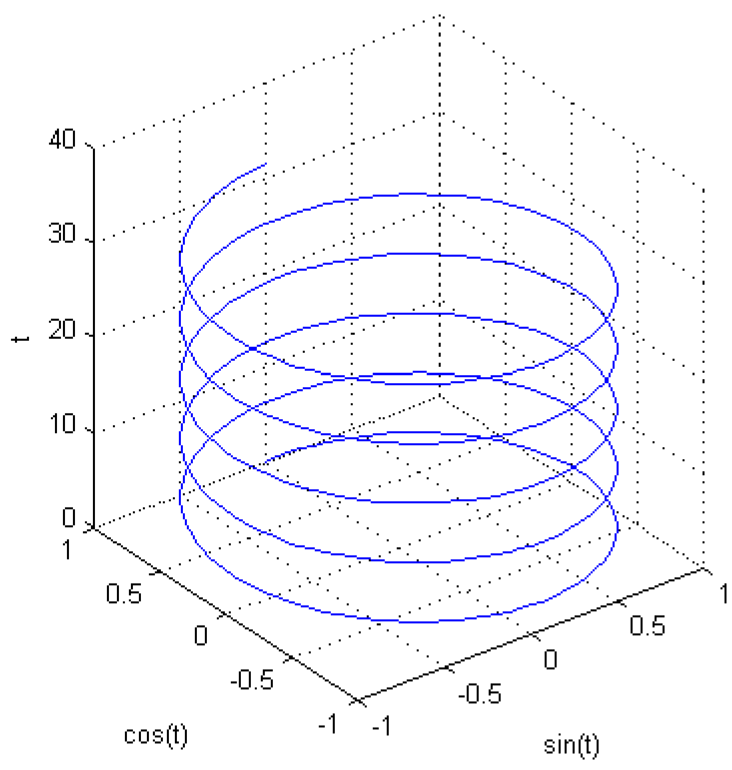
```
plot3(X1,Y1,Z1,X2,Y2,Z2,LineSpec,X3,Y3,Z3)
```

See LineSpec and plot for information on line types and markers.

Examples

Plot a three-dimensional helix.

```
t = 0:pi/50:10*pi;
plot3(sin(t),cos(t),t)
xlabel('sin(t)')
ylabel('cos(t)')
zlabel('t')
grid on
axis square
```



See Also

`axis` | `bar3` | `grid` | `line` | `LineSpec` | `lineseries properties` | `loglog` | `plot` | `scatter3` | `semilogx` | `semilogy` | `subplot`

plotbrowser

Purpose

Show or hide figure **Plot Browser**

Syntax

```
plotbrowser('on')
plotbrowser('off')
plotbrowser
plotbrowser(figure_handle,...)
```

Description

`plotbrowser('on')` displays the Plot Browser on the current figure.

`plotbrowser('off')` hides the Plot Browser on the current figure.

`plotbrowser` toggles the visibility of the Plot Browser on the current figure. You can use `plotbrowser('toggle')` instead for the same functionality.

`plotbrowser(figure_handle,...)` shows or hides the Plot Browser on the figure specified by `figure_handle`.

Tips

If you call `plotbrowser` in a MATLAB program and subsequent lines depend on the Plot Browser being fully initialized, follow it by `drawnow` to ensure complete initialization.

Alternatives

To collectively enable **Plotting Tools**, use the large Plotting Tool icon on the figure toolbar. To collectively disable the **Plotting Tools**, use the smaller icon. Open or close the **Plot Browser** tool from the figure's **View** menu.

See Also

`plottools` | `figurepalette` | `propertyeditor`

Purpose

Interactively edit and annotate plots

Syntax

```
plotedit on
plotedit off
plotedit
plotedit(h)
plotedit('state')
plotedit(h,'state')
```

Description

`plotedit on` starts plot edit mode for the current figure, allowing you to use a graphical interface to annotate and edit plots easily. In plot edit mode, you can label axes, change line styles, and add text, line, and arrow annotations.

`plotedit off` ends plot mode for the current figure.

`plotedit` toggles the plot edit mode for the current figure.

`plotedit(h)` toggles the plot edit mode for the figure specified by figure handle `h`.

`plotedit('state')` specifies the `plotedit` state for the current figure. Values for `state` can be as shown.

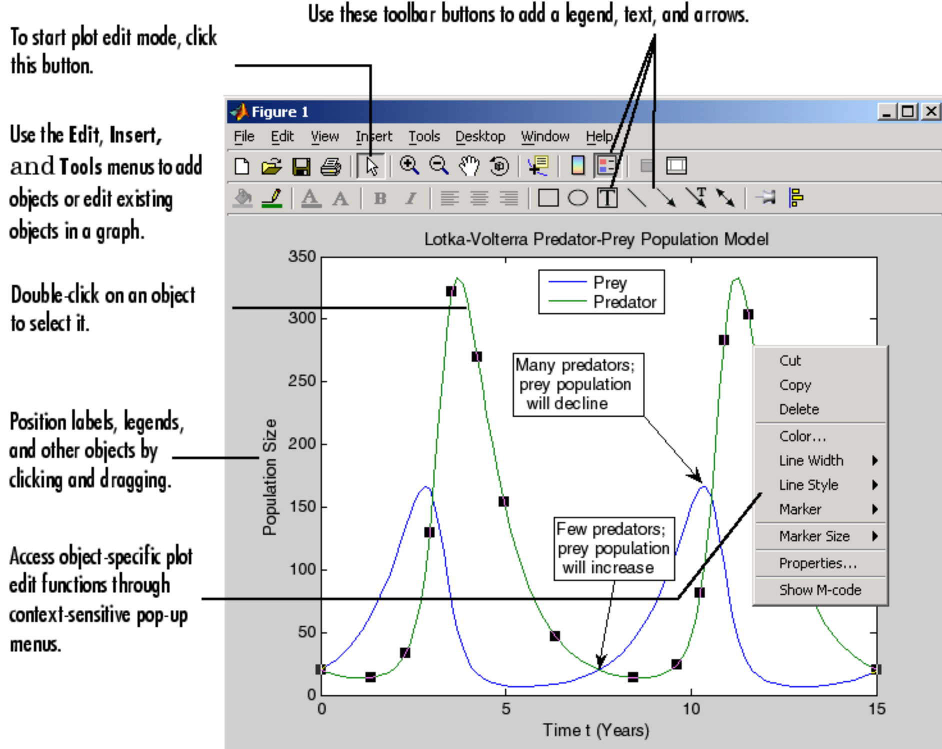
Value for state	Description
`on`	Starts plot edit mode
`off`	Ends plot edit mode
`showtoolsmenu`	Displays the **Tools** menu in the menu bar
`hidetoolsmenu`	Removes the **Tools** menu from the menu bar

Note `hidetoolsmenu` is intended for GUI developers who do not want the **Tools** menu to appear in applications that use the figure window.

`plotedit(h,'state')` specifies the `plotedit` `state` for figure handle `h`.

Tips

Plot Editing Mode Graphical Interface Components



Examples

Start plot edit mode for figure 2.

```
plotedit(2)
```

End plot edit mode for figure 2.

```
plotedit(2, 'off')
```

Hide the **Tools** menu for the current figure:

```
plotedit('hidetoolsmenu')
```

See Also

axes | line | open | plot | print | saveas | text | propedit

plotmatrix

Purpose

scatter plot matrix

Syntax

```
plotmatrix(X,Y)
plotmatrix(X)
plotmatrix(___,LineSpec)

[H,AX,BigAx,P,PAx] = plotmatrix(___)
```

Description

`plotmatrix(X,Y)` creates a matrix of subaxes containing scatter plots of the columns of `X` against the columns of `Y`. If `X` is p-by-n and `Y` is p-by-m, then `plotmatrix` produces an n-by-m matrix of subaxes.

`plotmatrix(X)` is the same as `plotmatrix(X,X)` except that the subaxes along the diagonal are replaced with histogram plots of the data in the corresponding column of `X`. For example, the subaxes along the diagonal in the ith column is replaced by `hist(X(:,i))`.

`plotmatrix(___,LineSpec)` specifies the line style, marker symbol, and color for the scatter plots. The option `LineSpec` can be preceded by any of the input argument combinations in the previous syntaxes.

`[H,AX,BigAx,P,PAx] = plotmatrix(___)` returns handles to the graphic objects created as follows:

- `H` – Matrix of handles to the scattergroup objects created
- `AX` – Matrix of handles to the individual subaxes
- `BigAx` – Handle to the big axes that frames the subaxes
- `P` – Vector of handles for the patch objects that create the histogram plots
- `PAx` – Vector of handles to the invisible histogram axes

`BigAx` is left as the current axes (`gca`) so that a subsequent `title`, `xlabel`, or `ylabel` command will center text with respect to the big axes.

Input Arguments

X - Data to display

matrix

Data to display, specified as a matrix.

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Y - Data to plot against X

matrix

Data to plot against `X`, specified as a matrix.

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

LineSpec - Line style, marker symbol, and color for scatter plots

string

Line style, marker symbol, and color for the scatter plots, specified as a string. For more information on line style, marker symbol, and color options see `LineSpec`.

Example: `':*r'`

Data Types

`char`

Output Arguments

H - Object handles

matrix

Object handles, returned as a matrix. This is a unique identifier, which you can use to query and modify the properties of a specific object.

AX - Subaxes handles

matrix

Subaxes handles, returned as a matrix. This is a unique identifier, which you can use to query and modify the properties of a specific subaxes.

BigAx - Big axes handle

scalar

Big axes handle, returned as a scalar. This is a unique identifier, which you can use to query and modify properties of the big axes. `BigAx` is left as the current axes (`gca`) so that a subsequent `title`, `xlabel`, or `ylabel` command will center text with respect to the big axes.

P - Patch object handles

vector | []

Patch object handles, returned as a vector or `[]`. If histogram plots are created, then `P` is returned as a vector of patch object handles for the histogram plots. These area unique identifiers, which you can use to query and modify the properties of a specific patch object. If no histogram plots are created, then `P` is returned as empty brackets.

PAx - Handle to invisible histogram axes

vector | []

Handle to invisible histogram axes, returned as a vector or `[]`. If histogram plots are created, then `PAx` is returned as a vector of histogram axes handles. These area unique identifiers, which you can use to query and modify the properties of a specific axes, such as the axes scale. If no histogram plots are created, then `PAx` is returned as empty brackets.

Examples

Create Scatter Plot Matrix with Two Matrix Inputs

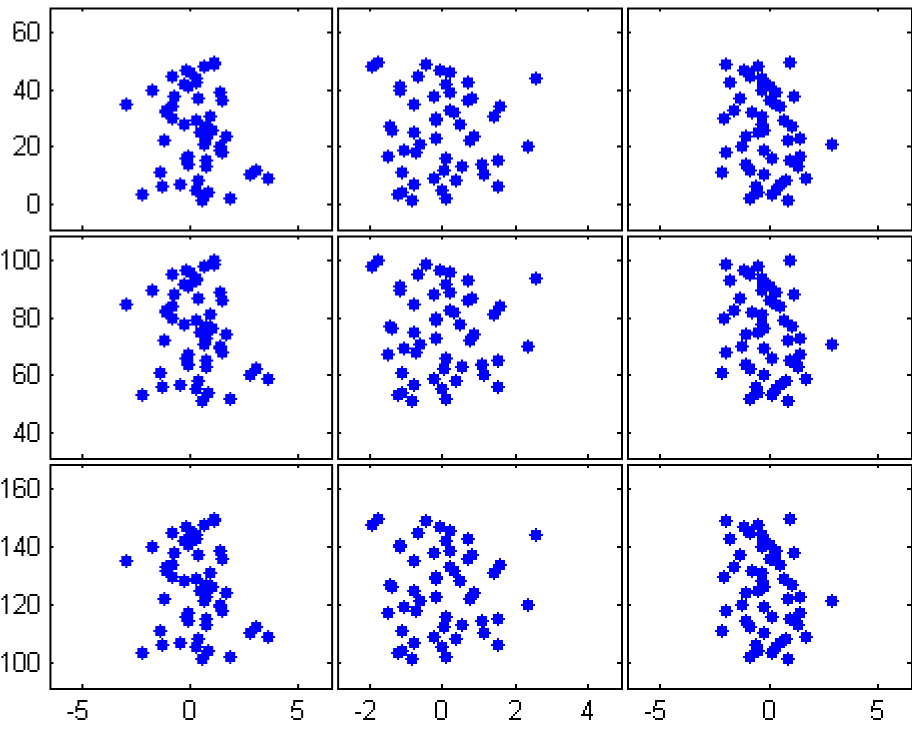
Initialize the random-number generator to make the output of `randn` repeatable. Set up `X` as a matrix of normally distributed pseudorandom data and `Y` as a matrix of integer values.

```
rng(0,'twister');
X = randn(50,3);
```

```
Y = reshape(1:150,50,3);
```

Create a scatter plot matrix of the columns of `X` against the columns of `Y`.

```
figure
plotmatrix(X,Y)
```



The subaxes in the *i*th row, *j*th column of the matrix is a scatter plot of the *i*th column of `Y` against the *j*th column of `X`.

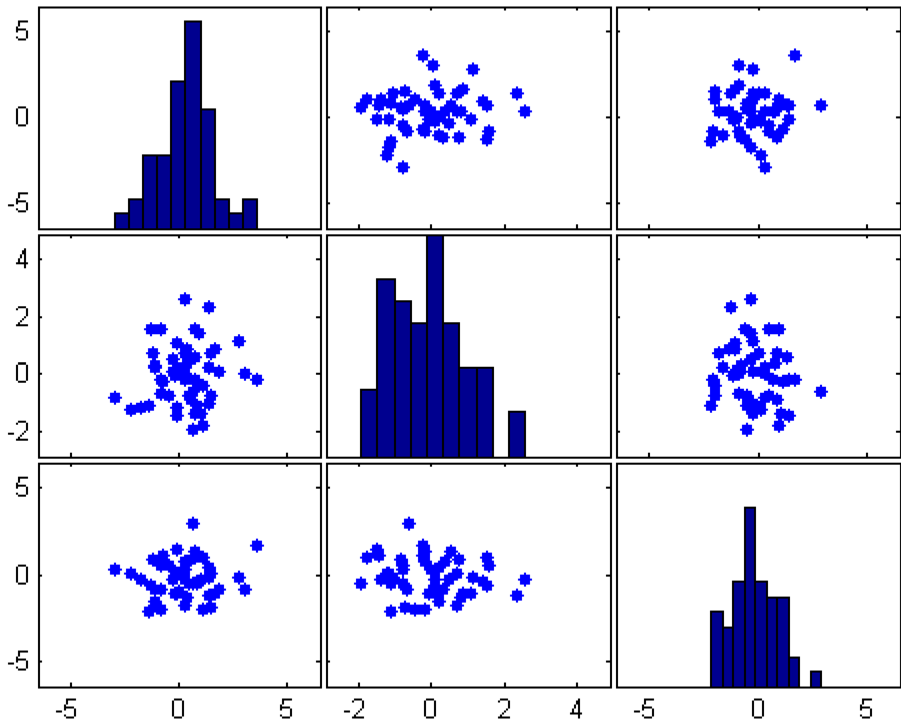
Create Scatter Plot Matrix with One Matrix Input

Initialize the random-number generator to make the output of `randn` repeatable and generate a matrix of normally distributed pseudorandom data

```
rng(0,'twister');
X = randn(50,3);
```

Create a scatter plot matrix.

```
figure
plotmatrix(X)
```



The subaxes in the *i*th row, *j*th column of the matrix is a scatter plot of the *i*th column of `X` against the *j*th column of `X`. Along the diagonal, `plotmatrix` creates a histogram plot of each column of `X`.

Specify Marker Type and Color

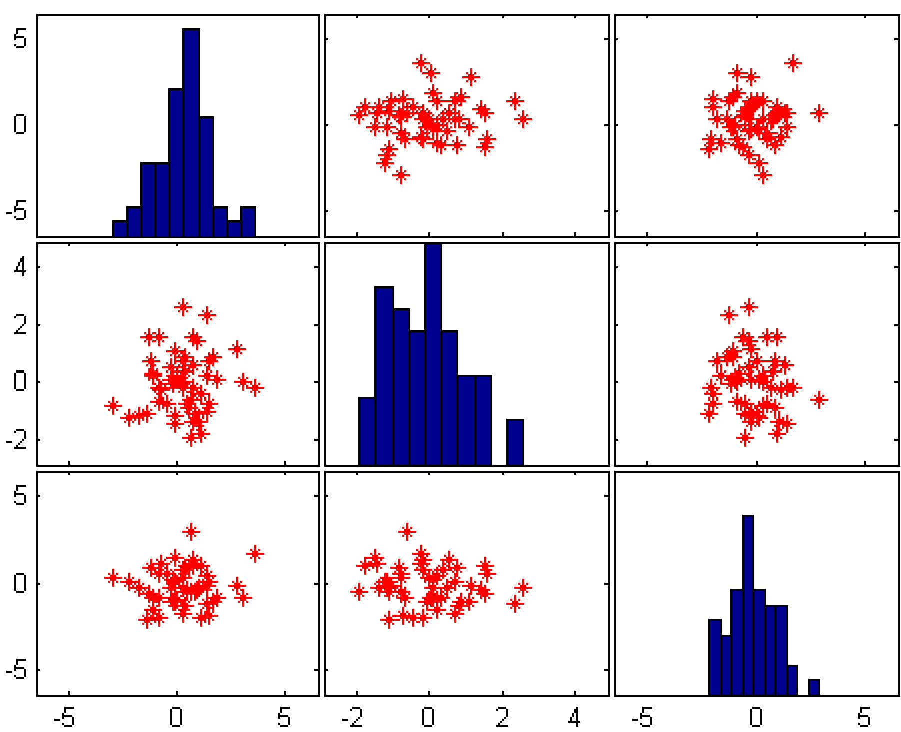
Initialize the random-number generator to make the output of `randn` repeatable. Generate a matrix of normally distributed pseudorandom data.

```
rng(0,'twister');
```

```
X = randn(50,3);
```

Create a scatter plot matrix and specify the marker type and the color for the scatter plots.

```
figure
plotmatrix(X,'*r')
```



`LineSpec` sets properties for the scatter plots. To set properties for the histogram plots, use the patch object handles.

Set Figure Properties Using Handles

Initialize the random-number generator to make the output of `randn` repeatable. Generate a matrix of normally distributed pseudorandom data.

```
rng(0,'twister');
X = randn(50,3);
```

Create a scatter plot matrix and get the handles to the objects created and the axes.

```
figure
[H,AX,BigAx,P,PAx] = plotmatrix(X);
```

To set properties for the scatter plot objects, use the handles in `H`. To set properties for the histogram patch objects, use the handles in `P`. To set axes properties, use the axes handles, `AX`, `BigAx`, and `PAx`.

Set the color and marker type for the scatter plot in the lower left corner.

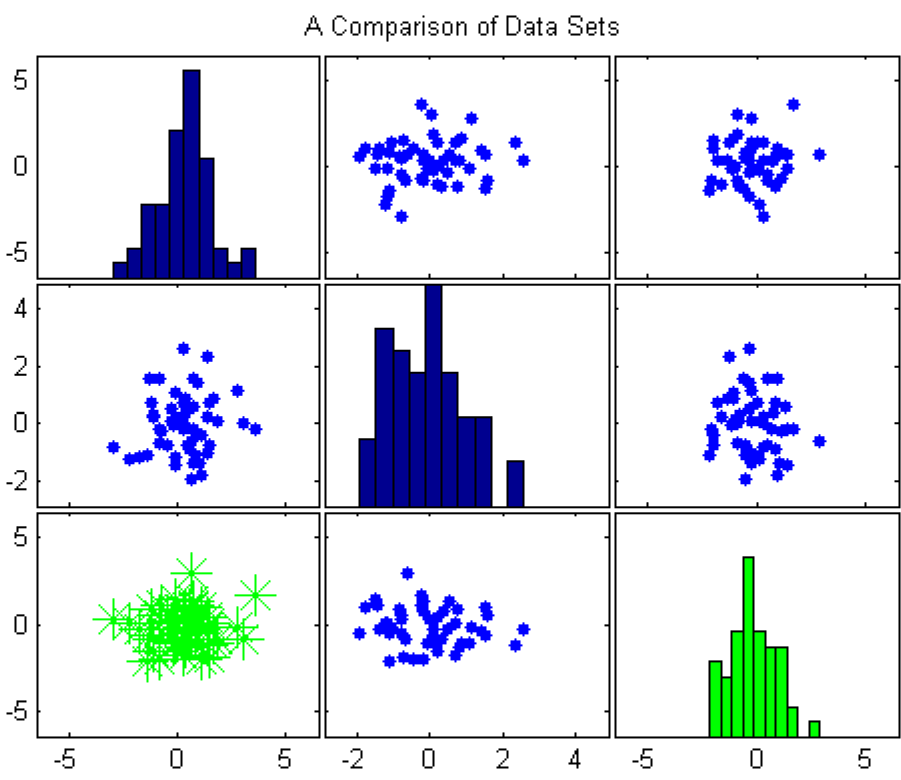
```
set(H(3),'Color','g','Marker','*');
```

Set the color for the histogram plot in the lower right corner.

```
set(P(3),'EdgeColor','k','FaceColor','g');
```

Use the `title` command to title the figure.

```
title(BigAx,'A Comparison of Data Sets');
```

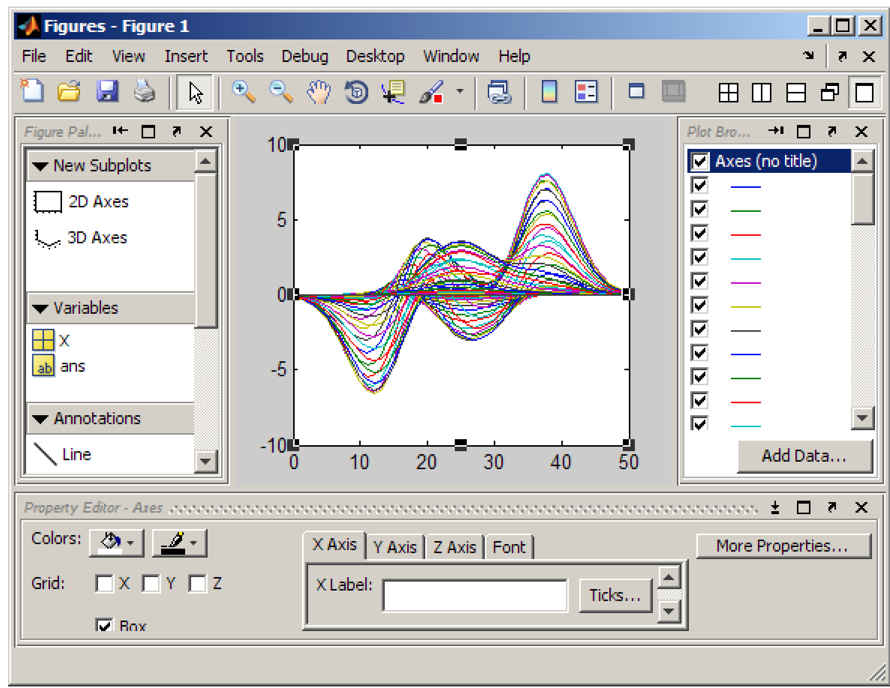


See Also

`scatter` | `scatter3`

Purpose

Show or hide plot tools



Syntax

```
plottools('on')
plottools('off')
plottools
plottools(figure_handle,...)
plottools(...,'tool')
```

Description

`plottools('on')` displays the Figure Palette, Plot Browser, and Property Editor on the current figure, configured as you last used them.

`plottools('off')` hides the Figure Palette, Plot Browser, and Property Editor on the current figure.

`plottools` with no arguments, is the same as `plottools('on')`

`plottools(figure_handle,...)` displays or hides the plot tools on the specified figure instead of on the current figure.

`plottools(...,'tool')` operates on the specified tool only. *`tool`* can be one of the following strings:

- `figurepalette`
- `plotbrowser`
- `propertyeditor`

Note The first time you open the plotting tools, all three of them appear, grouped around the current figure as shown above. If you close, move, or undock any of the tools, MATLAB remembers the configuration you left them in and restores it when you invoke the tools for subsequent figures, both within and across MATLAB sessions.

Alternatives

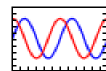
Click the larger **Plotting Tools** icon on the figure toolbar to collectively enable plotting tools, and the smaller icon to collectively disable them. Individually select the **Figure Palette**, **Plot Browser**, and **Property Editor** tools from the figure's **View** menu. For details, see "Plotting Tools — Interactive Plotting".

See Also

`figurepalette` | `plotbrowser` | `propertyeditor`

Purpose

2-D line plots with y-axes on both left and right side



Syntax

```
plotyy(X1,Y1,X2,Y2)
plotyy(X1,Y1,X2,Y2,function)
plotyy(X1,Y1,X2,Y2,'function1','function2')
[AX,H1,H2] = plotyy(...)
```

Description

`plotyy(X1,Y1,X2,Y2)` plots `X1` versus `Y1` with *y*-axis labeling on the left and plots `X2` versus `Y2` with *y*-axis labeling on the right.

`plotyy(X1,Y1,X2,Y2,function)` uses the specified plotting function to produce the graph.

`function` can be either a function handle or a string specifying `plot`, `semilogx`, `semilogy`, `loglog`, `stem`, or any MATLAB function that accepts the syntax

```
h = function(x,y)
```

For example,

```
plotyy(x1,y1,x2,y2,@loglog) % function handle
plotyy(x1,y1,x2,y2,'loglog') % string
```

Function handles enable you to access user-defined local functions and can provide other advantages. See @ for more information on using function handles.

`plotyy(X1,Y1,X2,Y2,'function1','function2')` uses `function1(X1,Y1)` to plot the data for the left axis and `function2(X2,Y2)` to plot the data for the right axis.

`[AX,H1,H2] = plotyy(...)` returns the handles of the two axes created in `AX` and the handles of the graphics objects from each plot in `H1` and `H2`. `AX(1)` is the left axes and `AX(2)` is the right axes.

Examples

This example graphs two mathematical functions using `plot` as the plotting function. The two *y*-axes enable you to display both sets of data on one graph even though relative values of the data are quite different.

```
figure
x = 0:0.01:20;
y1 = 200*exp(-0.05*x).*sin(x);
y2 = 0.8*exp(-0.5*x).*sin(10*x);
[AX,H1,H2] = plotyy(x,y1,x,y2,'plot');
```

You can use the handles returned by `plotyy` to label the axes and set the line styles used for plotting. With the axes handles you can specify the `YLabel` properties of the left- and right-side *y*-axis:

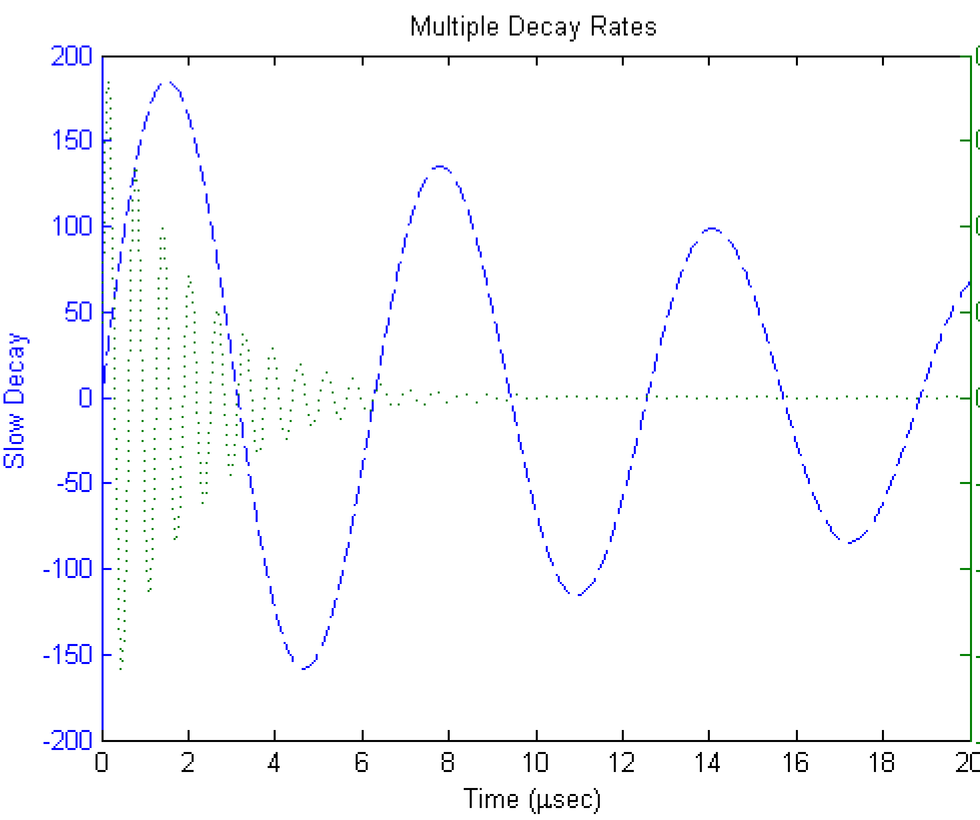
```
set(get(AX(1),'Ylabel'),'String','Slow Decay')
set(get(AX(2),'Ylabel'),'String','Fast Decay')
```

Use the `xlabel` and `title` commands to label the *x*-axis and add a title:

```
xlabel('Time (\musec)')
title('Multiple Decay Rates')
```

Use the line handles to set the `LineStyle` properties of the left- and right-side plots:

```
set(H1,'LineStyle','--')
set(H2,'LineStyle',':')
```



See Also

`plot` | `linkaxes` | `linkprop` | `loglog` | `semilogx` | `semilogy` | `XAxisLocation` | `YAxisLocation`

How To

- "Plotting with Two Y-Axes"
- "Using Multiple X- and Y-Axes"

plus

Purpose

Plus

Syntax

```
c = a+b
c = plus(a,b)
```

Description

`c = a+b` adds arrays `a` and `b` and returns the result in `c`. Inputs `a` and `b` must have the same size unless one is a scalar value (1-by-1 matrix). You can add a scalar value to any other value.

`c = plus(a,b)` is called for the syntax `a + b` when `a` or `b` is an object.

See Also

`minus`

Purpose

(Will be removed) Simplex containing specified location

Note `pointLocation(DelaunayTri)` will be removed in a future release. Use `pointLocation(delaunayTriangulation)` instead.

`DelaunayTri` will be removed in a future release. Use `delaunayTriangulation` instead.

Syntax

```
SI = pointLocation(DT,QX)
SI = pointLocation(DT,QX,QY)
SI = pointLocation(DT,QX,QY,QZ)
[SI, BC] = pointLocation(DT,...)
```

Description

`SI = pointLocation(DT,QX)` returns the indices `SI` of the enclosing simplex (triangle/tetrahedron) for each query point location in `QX`. The enclosing simplex for point `QX(k,:)` is `SI(k)`. `pointLocation` returns `NaN` for all points outside the convex hull.

`SI = pointLocation(DT,QX,QY)` and `SI = pointLocation(DT,QX,QY,QZ)` allow the query point locations to be specified in alternative column vector format when working in 2-D and 3-D.

`[SI, BC] = pointLocation(DT,...)` returns the barycentric coordinates `BC`.

Input Arguments

`DT`	Delaunay triangulation.
`QX`	Matrix of size `mpts`-by-`ndim`, `mpts` being the number of query points.

Output Arguments

`SI`	Column vector of length `mpts` containing the indices of the enclosing simplex for each query point. `mpts` is the number of query points.
`BC`	`BC` is a `mpts`-by-`ndim` matrix, each row `BC(i,:)` represents the barycentric coordinates of `QX(i,:)` with respect to the enclosing simplex `SI(i)`.

Examples

Example 1

Create a 2-D Delaunay triangulation:

```
X = rand(10,2);
dt = DelaunayTri(X);
```

Find the triangles that contain specified query points:

```
qrypts = [0.25 0.25; 0.5 0.5];
triids = pointLocation(dt, qrypts)
```

Example 2

Create a 3-D Delaunay triangulation:

```
x = rand(10,1);
y = rand(10,1);
z = rand(10,1);
dt = DelaunayTri(x,y,z);
```

Find the triangles that contain specified query points and evaluate the barycentric coordinates:

```
qrypts = [0.25 0.25 0.25; 0.5 0.5 0.5];
[tetids, bcs] = pointLocation(dt, qrypts)
```

See Also

`nearestNeighbor` | `delaunayTriangulation` | `triangulation`

Purpose

Transform polar or cylindrical coordinates to Cartesian

Syntax

```
[X,Y] = pol2cart(THETA,RHO)
[X,Y,Z] = pol2cart(THETA,RHO,Z)
```

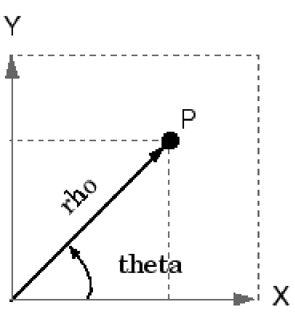
Description

`[X,Y] = pol2cart(THETA,RHO)` transforms the polar coordinate data stored in corresponding elements of `THETA` and `RHO` to two-dimensional Cartesian, or *xy*, coordinates. The arrays `THETA` and `RHO` must be the same size (or either can be scalar). The values in `THETA` must be in radians.

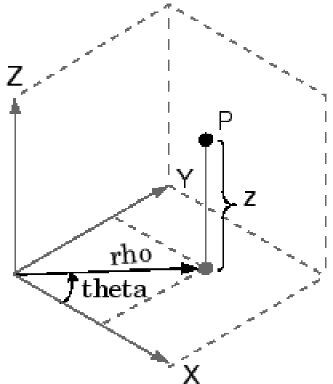
`[X,Y,Z] = pol2cart(THETA,RHO,Z)` transforms the cylindrical coordinate data stored in corresponding elements of `THETA`, `RHO`, and `Z` to three-dimensional Cartesian, or *xyz* coordinates. The arrays `THETA`, `RHO`, and `Z` must be the same size (or any can be scalar). The values in `THETA` must be in radians.

Algorithms

The mapping from polar and cylindrical coordinates to Cartesian coordinates is:



Polar to Cartesian Mapping
```
x = rho*cos(theta)
y = rho*sin(theta)
```



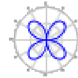
Polar to Cartesian Mapping
```
x = rho*cos(theta)
y = rho*sin(theta)
z = z
```

See Also

`cart2pol` | `cart2sph` | `sph2cart`

Purpose

Polar coordinate plot



Syntax

```
polar(theta,rho)
polar(theta,rho,LineSpec)
polar(axes_handle,...)
h = polar(...)
```

Description

The `polar` function accepts polar coordinates, plots them in a Cartesian plane, and draws the polar grid on the plane.

`polar(theta,rho)` creates a polar coordinate plot of the angle `theta` versus the radius `rho`. `theta` is the angle from the *x*-axis to the radius vector specified in radians; `rho` is the length of the radius vector specified in dataspace units.

`polar(theta,rho,LineSpec)` `LineSpec` specifies the line type, plot symbol, and color for the lines drawn in the polar plot.

`polar(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = polar(...)` returns the handle of a line object in `h`.

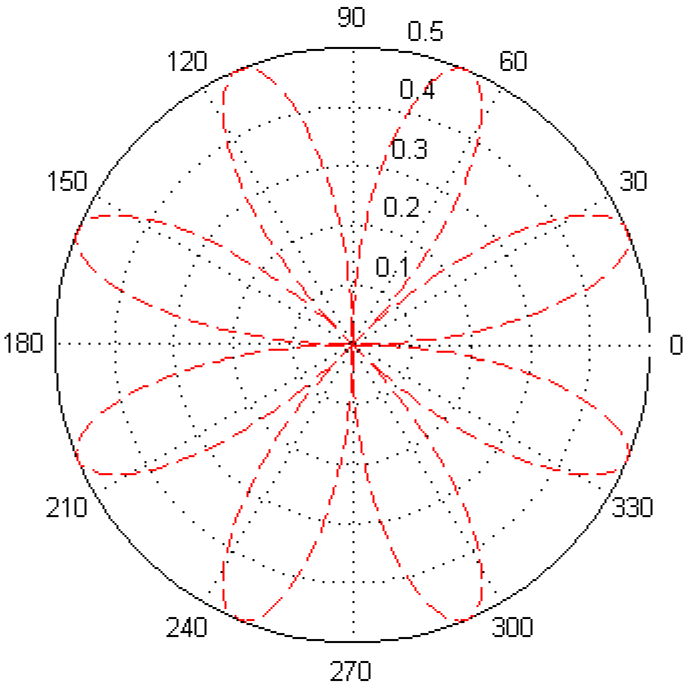
Tips

Negative `r` values reflect through the origin, rotating by `pi` (since `(theta,r)` transforms to `(r*cos(theta), r*sin(theta))`). If you want different behavior, you can manipulate `r` prior to plotting. For example, you can make `r` equal to `max(0,r)` or `abs(r)`.

Examples

Create a simple polar plot using a dashed red line:

```
figure
t = 0:.01:2*pi;
polar(t,sin(2*t).*cos(2*t),'--r')
```



See Also

`cart2pol` | `compass` | `LineSpec` | `plot` | `pol2cart` | `rose`

Purpose

Polynomial with specified roots

Syntax

```
p = poly(A)
p = poly(r)
```

Description

`p = poly(A)` where `A` is an `n`-by-`n` matrix returns an `n+1` element row vector whose elements are the coefficients of the characteristic polynomial, $\det(\lambda I - A)$. The coefficients are ordered in descending powers: if a vector `c` has `n+1` components, the polynomial it represents is $c_1 s^n + \ldots + c_n s + c_n + 1$

`p = poly(r)` where `r` is a vector returns a row vector whose elements are the coefficients of the polynomial whose roots are the elements of `r`.

Tips

Note the relationship of this command to

```
r = roots(p)
```

which returns a column vector whose elements are the roots of the polynomial specified by the coefficients row vector `p`. For vectors, `roots` and `poly` are inverse functions of each other, up to ordering, scaling, and roundoff error.

Examples

MATLAB displays polynomials as row vectors containing the coefficients ordered by descending powers. The characteristic equation of the matrix

```
A =

     1     2     3
     4     5     6
     7     8     0
```

is returned in a row vector by `poly`:

```
p = poly(A)

p =
     1     -6     -72     -27
```

The roots of this polynomial (eigenvalues of matrix `A`) are returned in a column vector by `roots`:

```
r = roots(p)

r =

    12.1229
    -5.7345
    -0.3884
```

Algorithms

The algorithms employed for `poly` and `roots` illustrate an interesting aspect of the modern approach to eigenvalue computation. `poly(A)` generates the characteristic polynomial of `A`, and `roots(poly(A))` finds the roots of that polynomial, which are the eigenvalues of `A`. But both `poly` and `roots` use `eig`, which is based on similarity transformations. The classical approach, which characterizes eigenvalues as roots of the characteristic polynomial, is actually reversed.

If `A` is an `n`-by-`n` matrix, `poly(A)` produces the coefficients `c(1)` through `c(n+1)`, with `c(1) = 1`, in

$$\det(\lambda I - A) = c_1\lambda^n + \ldots + c_n\lambda + c_{n+1}$$

The algorithm is

```
z = eig(A);
c = zeros(n+1,1); c(1) = 1;
for j = 1:n
    c(2:j+1) = c(2:j+1)-z(j)*c(1:j);
end
```

This recursion is easily derived by expanding the product.

$$(\lambda - \lambda_1)(\lambda - \lambda_2)\ldots(\lambda - \lambda_n)$$

It is possible to prove that `poly(A)` produces the coefficients in the characteristic polynomial of a matrix within roundoff error of `A`. This is

true even if the eigenvalues of `A` are badly conditioned. The traditional algorithms for obtaining the characteristic polynomial, which do not use the eigenvalues, do not have such satisfactory numerical properties.

See Also

`conv` | `polyval` | `residue` | `roots`

polyarea

Purpose

Area of polygon

Syntax

```
A = polyarea(X,Y)
A = polyarea(X,Y,dim)
```

Description

`A = polyarea(X,Y)` returns the area of the polygon specified by the vertices in the vectors `X` and `Y`.

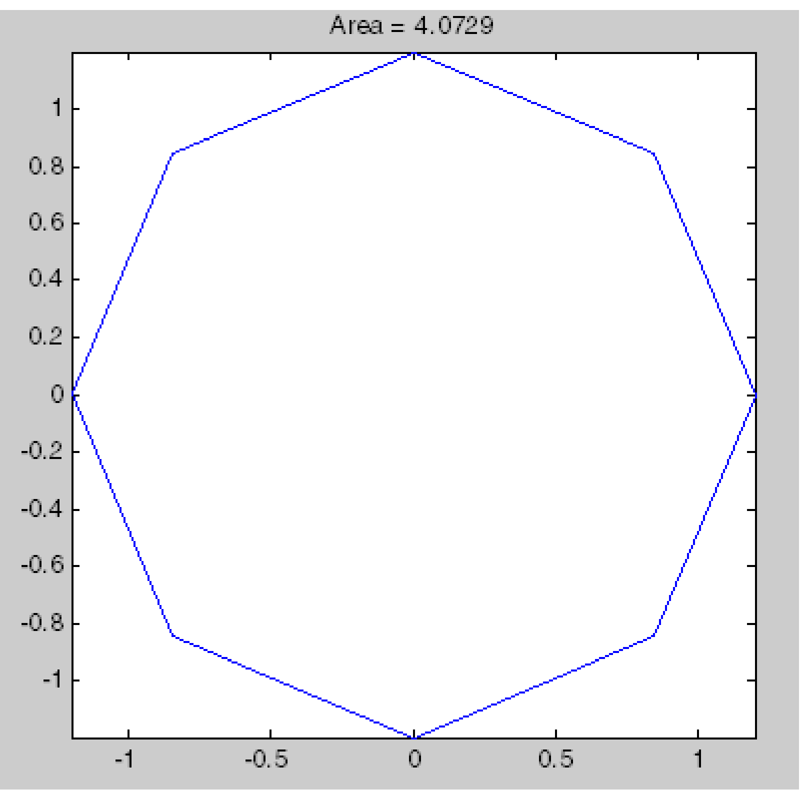
If `X` and `Y` are matrices of the same size, then `polyarea` returns the area of polygons defined by the columns `X` and `Y`.

If `X` and `Y` are multidimensional arrays, `polyarea` returns the area of the polygons in the first nonsingleton dimension of `X` and `Y`.

`A = polyarea(X,Y,dim)` operates along the dimension specified by scalar `dim`.

Examples

```
L = linspace(0,2.*pi,9); xv = 1.2*cos(L)';yv = 1.2*sin(L)';
xv = [xv ; xv(1)]; yv = [yv ; yv(1)];
A = polyarea(xv,yv);
plot(xv,yv); title(['Area = ' num2str(A)]); axis image
```



See Also

`convhull` | `inpolygon` | `rectint`

polyder

Purpose

Polynomial derivative

Syntax

```
k = polyder(p)
k = polyder(a,b)
[q,d] = polyder(b,a)
```

Description

The `polyder` function calculates the derivative of polynomials, polynomial products, and polynomial quotients. The operands `a`, `b`, and `p` are vectors whose elements are the coefficients of a polynomial in descending powers.

`k = polyder(p)` returns the derivative of the polynomial `p`.

`k = polyder(a,b)` returns the derivative of the product of the polynomials `a` and `b`.

`[q,d] = polyder(b,a)` returns the numerator `q` and denominator `d` of the derivative of the polynomial quotient `b/a`.

Examples

The derivative of the product

$$\left(3x^2+6x+9\right)\left(x^2+2x\right)$$

is obtained with

```
a = [3 6 9];
b = [1 2 0];
k = polyder(a,b)
k =
    12    36    42    18
```

This result represents the polynomial

$$12x^3+36x^2+42x+18$$

See Also

`conv` | `deconv`

Purpose

Polynomial eigenvalue problem

Syntax

```
[X,e] = polyeig(A0,A1,...Ap)
e = polyeig(A0,A1,..,Ap)
[X, e, s] = polyeig(A0,A1,..,AP)
```

Description

`[X,e] = polyeig(A0,A1,...Ap)` solves the polynomial eigenvalue problem of degree `p`

$$\left(A_0 + \lambda A_1 + \ldots + \lambda^P A_p\right)x = 0$$

where polynomial degree `p` is a non-negative integer, and `A0,A1,...Ap` are input matrices of order `n`. The output consists of a matrix `X` of size `n`-by-`n*p` whose columns are the eigenvectors, and a vector `e` of length `n*p` containing the eigenvalues.

If `lambda` is the `j`th eigenvalue in `e`, and `x` is the `j`th column of eigenvectors in `X`, then `(A0 + lambda*A1 + ... + lambda^p*Ap)*x` is approximately `0`.

`e = polyeig(A0,A1,..,Ap)` is a vector of length `n*p` whose elements are the eigenvalues of the polynomial eigenvalue problem.

`[X, e, s] = polyeig(A0,A1,..,AP)` also returns a vector `s` of length `p*n` containing condition numbers for the eigenvalues. At least one of `A0` and `AP` must be nonsingular. Large condition numbers imply that the problem is close to a problem with multiple eigenvalues.

Tips

Based on the values of `p` and `n`, `polyeig` handles several special cases:

- `p = 0`, or `polyeig(A)` is the standard eigenvalue problem: `eig(A)`.
- `p = 1`, or `polyeig(A,B)` is the generalized eigenvalue problem: `eig(A,-B)`.
- `n = 1`, or `polyeig(a0,a1,...ap)` for scalars `a0`, `a1` ..., `ap` is the standard polynomial problem: `roots([ap ... a1 a0])`.

If both A0 and Ap are singular the problem is potentially ill-posed. Theoretically, the solutions might not exist or might not be unique. Computationally, the computed solutions might be inaccurate. If one, but not both, of A0 and Ap is singular, the problem is well posed, but some of the eigenvalues might be zero or infinite.

Note that scaling A0,A1,..,Ap to have norm(Ai) roughly equal 1 may increase the accuracy of polyeig. In general, however, this cannot be achieved. (See Tisseur [3] for more detail.)

Algorithms

The polyeig function uses the QZ factorization to find intermediate results in the computation of generalized eigenvalues. It uses these intermediate results to determine if the eigenvalues are well-determined. See the descriptions of eig and qz for more on this.

References

[1] Dedieu, Jean-Pierre Dedieu and Francoise Tisseur, “Perturbation theory for homogeneous polynomial eigenvalue problems,” *Linear Algebra Appl.*, Vol. 358, pp. 71-94, 2003.

[2] Tisseur, Francoise and Karl Meerbergen, “The quadratic eigenvalue problem,” *SIAM Rev.*, Vol. 43, Number 2, pp. 235-286, 2001.

[3] Francoise Tisseur, “Backward error and condition of polynomial eigenvalue problems” *Linear Algebra Appl.*, Vol. 309, pp. 339-361, 2000.

See Also

condeig | eig | qz

Purpose

Polynomial curve fitting

Syntax

```
p = polyfit(x,y,n)
[p,S] = polyfit(x,y,n)
[p,S,mu] = polyfit(x,y,n)
```

Description

`p = polyfit(x,y,n)` finds the coefficients of a polynomial `p(x)` of degree `n` that fits the data, `p(x(i))` to `y(i)`, in a least squares sense. The result `p` is a row vector of length `n+1` containing the polynomial coefficients in descending powers:

$$p(x) = p_1 x^n + p_2 x^{n-1} + \ldots + p_n x + p_{n+1}.$$

`[p,S] = polyfit(x,y,n)` returns the polynomial coefficients `p` and a structure `S` for use with `polyval` to obtain error estimates or predictions. Structure `S` contains fields `R`, `df`, and `normr`, for the triangular factor from a QR decomposition of the Vandermonde matrix of `x`, the degrees of freedom, and the norm of the residuals, respectively. If the data `y` are random, an estimate of the covariance matrix of `p` is `(Rinv*Rinv')*normr^2/df`, where `Rinv` is the inverse of `R`. If the errors in the data `y` are independent normal with constant variance, `polyval` produces error bounds that contain at least 50% of the predictions.

`[p,S,mu] = polyfit(x,y,n)` finds the coefficients of a polynomial in

$$\hat{x} = \frac{x - \mu_1}{\mu_2}$$

where $\mu_1 = \mathrm{mean}(x)$ and $\mu_2 = \mathrm{std}(x)$. `mu` is the two-element vector $[\mu_1, \mu_2]$. This centering and scaling transformation improves the numerical properties of both the polynomial and the fitting algorithm.

Examples

This example involves fitting the error function, `erf(x)`, by a polynomial in `x`. This is a risky project because `erf(x)` is a bounded function, while polynomials are unbounded, so the fit might not be very good.

First generate a vector of `x` points, equally spaced in the interval [0, 2.5]; then evaluate `erf(x)` at those points.

```
x = (0: 0.1: 2.5)';
y = erf(x);
```

The coefficients in the approximating polynomial of degree 6 are

```
p = polyfit(x,y,6)

p =

 0.0084  -0.0983   0.4217   -0.7435  0.1471   1.1064  0.0004
```

There are seven coefficients, and the polynomial is

$$0.0084x^6 - 0.0983x^5 + 0.4217x^4 - 0.7435x^3 + 0.1471x^2 + 1.1064x + 0.0004$$

To see how good the fit is, evaluate the polynomial at the data points with:

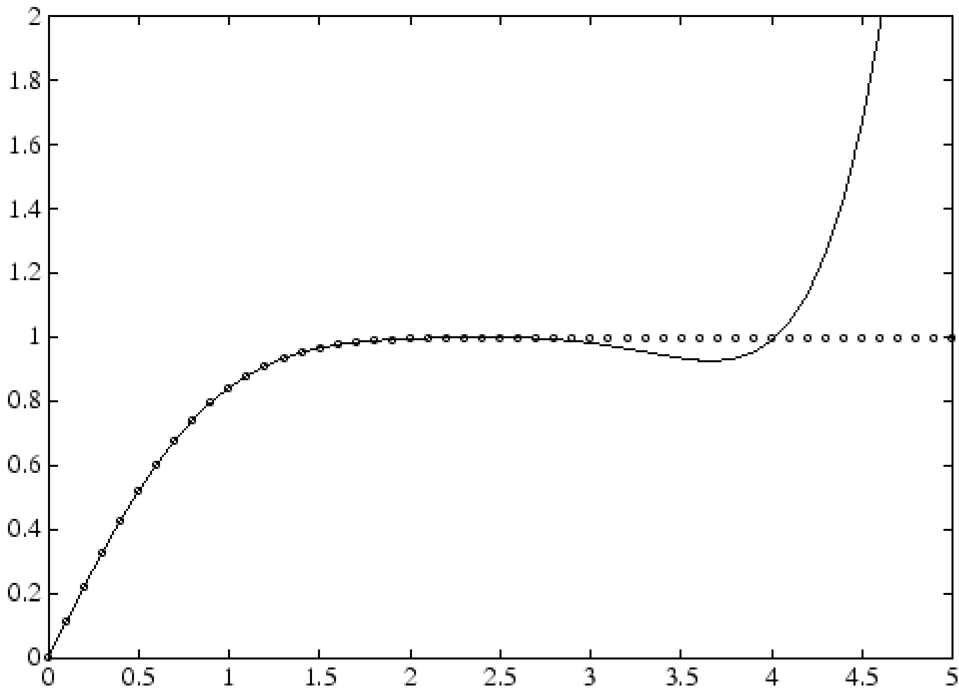
```
f = polyval(p,x);
```

A table showing the data, fit, and error is

```
 table = [x y f y-f]

 table =

    0          0          0.0004    -0.0004
    0.1000     0.1125     0.1119     0.0006
    0.2000     0.2227     0.2223     0.0004
    0.3000     0.3286     0.3287    -0.0001
    0.4000     0.4284     0.4288    -0.0004
    ...
    2.1000     0.9970     0.9969     0.0001
    2.2000     0.9981     0.9982    -0.0001
    2.3000     0.9989     0.9991    -0.0003
    2.4000     0.9993     0.9995    -0.0002
    2.5000     0.9996     0.9994     0.0002
```

On this interval, the interpolated values and the actual values agree fairly closely. Outside this interval, the extrapolated values quickly diverge from the actual data values, as the following plot demonstrates.

```
x = (0: 0.1: 5)';
y = erf(x);
f = polyval(p,x);
plot(x,y,'o',x,f,'-')
axis([0  5  0  2])
```



Algorithms

The `polyfit` MATLAB file forms the Vandermonde matrix, V, whose elements are powers of x. $v_{i,j} = x_i^{n-j}$

It then uses the backslash operator, \, to solve the least squares problem $Vp \cong y$.

See "Programmatic Fitting" for information about fitting different functions of x.

See Also

poly | polyval | roots | lscov | cov

Purpose

Integrate polynomial analytically

Syntax

```
polyint(p,k)
polyint(p)
```

Description

`polyint(p,k)` returns a polynomial representing the integral of polynomial `p`, using a scalar constant of integration `k`.

`polyint(p)` assumes a constant of integration `k=0`.

See Also

`polyder` | `polyval` | `polyvalm` | `polyfit`

polyval

Purpose

Polynomial evaluation

Syntax

```
y = polyval(p,x)
[y,delta] = polyval(p,x,S)
y = polyval(p,x,[],mu)
[y,delta] = polyval(p,x,S,mu)
```

Description

`y = polyval(p,x)` returns the value of a polynomial of degree `n` evaluated at `x`. The input argument `p` is a vector of length `n+1` whose elements are the coefficients in descending powers of the polynomial to be evaluated.

$$y = p_1 x^n + p_2 x^{n-1} + \ldots + p_n x + p_{n+1}$$

`x` can be a matrix or a vector. In either case, `polyval` evaluates `p` at each element of `x`.

`[y,delta] = polyval(p,x,S)` uses the optional output structure `S` generated by `polyfit` to generate error estimates `delta`. `delta` is an estimate of the standard deviation of the error in predicting a future observation at `x` by `p(x)`. If the coefficients in `p` are least squares estimates computed by `polyfit`, and the errors in the data input to `polyfit` are independent, normal, and have constant variance, then `y±delta` contains at least 50% of the predictions of future observations at `x`.

`y = polyval(p,x,[],mu)` or `[y,delta] = polyval(p,x,S,mu)` use $\hat{x} = (x - \mu_1)/\mu_2$ in place of `x`. In this equation, $\mu_1 = \text{mean}(x)$ and $\mu_2 = \text{std}(x)$. The centering and scaling parameters `mu` = $[\mu_1, \mu_2]$ are optional output computed by `polyfit`.

Tips

The `polyvalm(p,x)` function, with `x` a matrix, evaluates the polynomial in a matrix sense. See `polyvalm` for more information.

Examples

The polynomial $p(x) = 3x^2 + 2x + 1$ is evaluated at $x = 5$, 7, and 9 with

```
p = [3 2 1];
polyval(p,[5 7 9])
```

which results in

```
ans =

    86   162   262
```

For another example, see `polyfit`.

See Also

`polyfit` | `polyvalm` | `polyder` | `polyint`

polyvalm

Purpose

Matrix polynomial evaluation

Syntax

```
Y = polyvalm(p,X)
```

Description

`Y = polyvalm(p,X)` evaluates a polynomial in a matrix sense. This is the same as substituting matrix `X` in the polynomial `p`.

Polynomial `p` is a vector whose elements are the coefficients of a polynomial in descending powers, and `X` must be a square matrix.

Examples

The Pascal matrices are formed from Pascal's triangle of binomial coefficients. Here is the Pascal matrix of order 4.

```
X = pascal(4)
X =
     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20
```

Its characteristic polynomial can be generated with the `poly` function.

```
p = poly(X)
p =
     1     -29     72     -29     1
```

This represents the polynomial $x^4 - 29x^3 + 72x^2 - 29x + 1$.

Pascal matrices have the curious property that the vector of coefficients of the characteristic polynomial is palindromic; it is the same forward and backward.

Evaluating this polynomial at each element is not very interesting.

```
polyval(p,X)
ans =
    16        16        16        16
    16        15      -140      -563
    16      -140     -2549    -12089
```

```
    16     -563   -12089   -43779
```

But evaluating it in a matrix sense is interesting.

```
polyvalm(p,X)
ans =
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
```

The result is the zero matrix. This is an instance of the Cayley-Hamilton theorem: a matrix satisfies its own characteristic equation.

See Also

polyfit | polyval

pow2

Purpose

Base 2 power and scale floating-point numbers

Syntax

```
X = pow2(Y)
X = pow2(F,E)
```

Description

`X = pow2(Y)` returns an array `X` whose elements are 2 raised to the power `Y`.

`X = pow2(F,E)` computes $x = f * 2^e$ for corresponding elements of `F` and `E`. The result is computed quickly by simply adding `E` to the floating-point exponent of `F`. Arguments `F` and `E` are real and integer arrays, respectively.

Tips

This function corresponds to the ANSI C function `ldexp()` and the IEEE floating-point standard function `scalbn()`.

Examples

For IEEE arithmetic, the statement `X = pow2(F,E)` yields the values:

```
F          E        X
1/2        1        1
pi/4       2        pi
-3/4       2        -3
1/2        -51      eps
1-eps/2    1024     realmax
1/2        -1021    realmin
```

See Also

`log2` | `exp` | `hex2num` | `realmax` | `realmin` | `Arithmetic Operators ^ and .^`

Purpose

Array power

Syntax

```
Z = X.^Y
```

Description

`Z = X.^Y` denotes element-by-element powers. `X` and `Y` must have the same dimensions unless one is a scalar. A scalar is expanded to an array of the same size as the other input.

`C = power(A,B)` is called for the syntax `'A .^ B'` when `A` or `B` is an object.

Note that for a negative value `X` and a non-integer value `Y`, if the `abs(Y)` is less than one, the `power` function returns the complex roots. To obtain the remaining real roots, use the `nthroot` function.

See Also

`nthroot` | `realpow`

Purpose Evaluate piecewise polynomial

Syntax

```
v = ppval(pp,xx)
```

Description

`v = ppval(pp,xx)` returns the value of the piecewise polynomial *f*, contained in `pp`, at the entries of `xx`. You can construct `pp` using the functions `pchip`, `spline`, or the spline utility `mkpp`.

`v` is obtained by replacing each entry of `xx` by the value of *f* there. If *f* is scalar-valued, `v` is of the same size as `xx`. `xx` may be N-dimensional.

If `pp` was constructed by `pchip`, `spline`, or `mkpp` using the orientation of non-scalar function values specified for those functions, then:

If `f` is `[D1,..,Dr]`-valued, and `xx` is a vector of length `N`, then `V` has size `[D1,...,Dr, N]`, with `V(:,...,:,J)` the value of `f` at `xx(J)`.

If `f` is `[D1,..,Dr]`-valued, and `xx` has size `[N1,...,Ns]`, then `V` has size `[D1,...,Dr, N1,...,Ns]`, with `V(:,...,:, J1,...,Js)` the value of `f` at `xx(J1,...,Js)`.

Examples

Compare the result of integrating *cos*(*x*) between 0 and 10 to the result of integrating a piece-wise polynomial approximation of the same function.

```
a = 0; b = 10;
int1 = integral(@cos,a,b)

int1 =
   -0.5440
```

Create a piece-wise polynomial approximation of *cos*(*x*) and integrate over the same interval.

```
x = a:b;
y = cos(x);
pp = spline(x,y);
int2 = integral(@(x)ppval(pp,x),a,b)

int2 =
```

```
   -0.5485
```

See Also

mkpp | spline | unmkpp

prefdir

Purpose

Folder containing preferences, history, and layout files

Syntax

```
prefdir
folder = prefdir
folder = prefdir(1)
```

Description

`prefdir` returns the folder that contains

- Preferences for MATLAB and related products (`matlab.prf`)
- Command history file (`history.m`)
- MATLAB shortcuts (`shortcuts_2.xml`)
- MATLAB desktop layout files (`MATLABDesktop.xml` and `Your_Saved_LayoutMATLABLayout.xml`)
- Other related files

`folder = prefdir` assigns to `folder` the name of the folder containing preferences and related files.

`folder = prefdir(1)` creates a folder for preferences and related files if one does not exist. If the folder does exist, the name is assigned to `folder`.

Tips

- You must have write access to the preferences folder. Otherwise, MATLAB generates an error in the Command Window when you try to change preferences. This can happen if the folder is hidden, for example: `myname/.matlab/R2009a`.

Examples

View the location of the preferences folder:

```
prefdir
```

Make the preferences folder become the current folder:

```
cd(prefdir)
```

```
% Then, view the files for customizing MathWorks products:
dir
```

On Windows platforms, go directly to the preferences folder in Microsoft Windows Explorer

```
winopen(prefdir)
```

See Also

`preferences` | `getpref` | `setpref`

How To

- "Preferences Folder and Files MATLAB Uses When Multiple MATLAB Releases Are Installed"
- "Viewing Hidden Files and Folders"

Purpose Open Preferences dialog box

Syntax

```
preferences
```

Description `preferences` displays the Preferences dialog box, from which you can make changes to options for MATLAB and related products.

See Also `prefdir`

How To

- "Preferences"

Purpose

Prime numbers less than or equal to input value

Syntax

```
p = primes(n)
```

Description

`p = primes(n)` returns a row vector containing all the prime numbers less than or equal to `n`. The data type of `p` is the same as that of `n`.

Input Arguments

n - Input value

scalar, real integer value

Input value, specified as a scalar that is a real integer value.

Example: `10`

Example: `int16(32)`

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Examples

Primes Less Than or Equal to 25

```
p = primes(25)

p =

     2     3     5     7    11    13    17    19    23
```

Primes Less Than or Equal to an Unsigned Integer

```
n = uint16(12);
p = primes(n)

p =

     2     3     5     7    11
```

See Also

`factor` | `isprime`

Purpose

Print figure or save to file

Contents

Syntax

"Description" on page 1-4016

"Printer Drivers" on page 1-4018

"Graphics Format Files" on page 1-4022

"Printing Options" on page 1-4026

"Paper Sizes" on page 1-4029

"Printing Tips" on page 1-4031

"Examples" on page 1-4034

"Alternatives" on page 1-4036

Syntax

```
print
print('argument1','argument2',...)
print(handle,'filename')
print argument1 argument2 ... argumentn
```

Description

`print` sends the contents of the current figure, including bitmap representations of any user interface controls, to the printer using the device and system printing command defined by `printopt`.

`print('`*`argument1`*`','`*`argument2`*`',...)` is the function form of `print`. It enables you to pass variables for any input arguments. This form is useful for passing file names and handles (for example, `print(`*`handle`*`,'`*`filename`*`')`. See "Batch Processing" on page 1-4035 for an example. Also see "Specifying the Figure to Print" on page 1-4034 for further examples.

Note Print input only takes string variables and handles as inputs. Strings passed in as cell array or struct inputs are not accepted.

`print` *`argument1 argument2 ... argumentn`* prints the figure using the specified arguments.

The following arguments apply to both the function and the command form:

Argument	Description
`handle`	Print the specified object.
`filename`	Direct the output to the PostScript file designated by `filename`. If `filename` does not include an extension, `print` appends an appropriate extension.
`-d`*`driver`*	Print the figure using the specified printer *`driver`*, (such as color PostScript). If you omit `-d`*`driver`*, `print` uses the default value stored in `printopt.m`. The table in "Printer Drivers" on page 1-4018 lists all supported device types.
`-d`*`format`*	Copy the figure to the system Clipboard (Microsoft Windows platforms only). To be valid, the *`format`* for this operation must be either `-dmeta` (Windows Enhanced Metafile) or `-dbitmap` (Windows Bitmap).
`-d`*`format`* `filename`	Export the figure to the specified file using the specified graphics *`format`* (such as TIFF). The table of "Graphics Format Files" on page 1-4022 lists all supported graphics file formats.
`-s`*`modelname`*	Print the current Simulink model *`modelname`*.
`-options`	Specify print options that modify the action of the `print` command. (For example, the `-noui` option suppresses printing of user interface controls.) "Printing Options" on page 1-4026 lists available options.

Printer Drivers

The following table shows the more widely used printer drivers supported by MATLAB software. If you do not specify a driver, the default setting shown in the previous table is used. For a list of all supported printer drivers, type `print -d` at the MATLAB prompt. Some things to remember:

- As indicated in "Description" on page 1-4016 the `-d` switch specifies a printer driver or a graphics file format:
  - Specifying a printer driver without a file name or printer name (the `-P` option) sends the output formatted by the specified driver to your default printer, which may not be what you want to do.

    Note On Windows systems, when you use the `-P` option to identify a printer to use, if you specify any driver other than `-dwin` or `-dwinc`, MATLAB writes the output to a file with an appropriate extension but does not send it to the printer. You can then copy that file to a printer.

  - Specifying a `-dmeta` or a `-dbitmap` graphics format without a file name places the graphic on the system Clipboard, if possible (Windows platforms only).
  - Specifying any other graphics format without a file name creates a file in the current folder with a name such as `figureN.`*`fmt`*, where `N` is 1, 2, 3, ... and *`fmt`* indicates the format type, for example, `eps` or `png`.
- Several drivers come from a product called Ghostscript, which is shipped with MATLAB software. The last column indicates when Ghostscript is used.
- Not all drivers are supported on all platforms. Non support is noted in the first column of the table.
- If you specify a particular printer with the `-P` option and do not specify a driver, a default driver for that printer is selected, either by the operating system or by MATLAB , depending on the platform:

- On Windows, the driver associated with this particular printing device is used.
- On Macintosh and UNIX platforms, the driver specified in `printopt.m` is used

See Selecting the Printer in the Graphics documentation for more information.

Note Support for some print options will be removed in a future release. The affected formats have an asterisk (*) next to the option string in the following table. The asterisks provide a link to the Web site which supplies a form for users to give feedback about these changes.

Printer Driver	Print Command Option String	Ghostscript
Canon® BubbleJet BJ10e	`-dbj10e` *	Yes
Canon BubbleJet BJ200 color	`-dbj200` *	Yes
Canon Color BubbleJet BJC-70/BJC-600/BJC-4000	`-dbjc600` *	Yes
Canon Color BubbleJet BJC-800	`-dbjc800` *	Yes
Epson® and compatible 9- or 24-pin dot matrix print drivers	`-depson` *	Yes
Epson and compatible 9-pin with interleaved lines (triple resolution)	`-deps9high` *	Yes

Printer Driver	Print Command Option String	Ghostscript
Epson LQ-2550 and compatible; color (not supported on HP-700)	-depsonc *	Yes
Fujitsu® 3400/2400/1200	-depsonc *	Yes
HP® DesignJet 650C color (not supported on Windows)	-ddnj650c *	Yes
HP DeskJet 500	-ddjet500 *	Yes
HP DeskJet 500C (creates black and white output)	-dcdjmono *	Yes
HP DeskJet 500C (with 24 bit/pixel color and high-quality Floyd-Steinberg color dithering) (not supported on Windows)	-dcdjcolor *	Yes
HP DeskJet 500C/540C color (not supported on Windows)	-dcdj500 *	Yes
HP Deskjet 550C color (not supported on Windows)	-dcdj550 *	Yes
HP DeskJet and **DeskJet Plus**	-ddeskjet *	Yes
HP LaserJet	-dlaserjet *	Yes
HP LaserJet+	-dljetplus *	Yes
HP LaserJet IIP	-dljet2p *	Yes

Printer Driver	Print Command Option String	Ghostscript
HP LaserJet III	`-dljet3` *	Yes
HP LaserJet 4, 5L and 5P	`-dljet4` *	Yes
HP LaserJet 5 and 6	`-dpxlmono` *	Yes
HP PaintJet color	`-dpaintjet` *	Yes
HP PaintJet XL color	`-dpjxl` *	Yes
HP PaintJet XL color	`-dpjetxl` *	Yes
HP PaintJet XL300 color (not supported on Windows)	`-dpjxl300` *	Yes
HPGL for HP 7475A and other compatible plotters. (Renderer cannot be set to Z-buffer.)	`-dhpgl` *	No
IBM 9-pin Proprinter	`-dibmpro` *	Yes
PostScript black and white	`-dps`	No
PostScript color	`-dpsc`	No
PostScript Level 2 black and white	`-dps2`	No
PostScript Level 2 color	`-dpsc2`	No
Windows color (Windows only)	`-dwinc`	No
Windows monochrome (Windows only)	`-dwin`	No

Tip Generally, Level 2 PostScript files are smaller and are rendered more quickly when printing than Level 1 PostScript files. However, not all PostScript printers support Level 2, so determine the capabilities of your printer before using those drivers. Level 2 PostScript printing is the default for UNIX platforms. You can change this default by editing the `printopt.m` file. Likewise, if you want color PostScript printing to be the default instead of black-and-white PostScript printing, edit the line in the `printopt.m` file that reads `dev = '-dps2';` to be `dev = '-dpsc2';`.

Graphics Format Files

To save your figure as a graphics format file, specify a format switch and file name. To set the resolution of the output file for a built-in MATLAB format, use the `-r` switch. (For example, `-r300` sets the output resolution to 300 dots per inch.) The `-r` switch is also supported for Windows Enhanced Metafiles, JPEG, TIFF and PNG files, but is not supported for Ghostscript raster formats. For more information, see “Printing and Exporting without a Display” on page 1-4025 and “Resolution Considerations” on page 1-4028.

Note When you print to a file, the file name must have fewer than 128 characters, including path name. When you print to a file in your current folder, the filename must have fewer than 126 characters, because MATLAB places `'./'` or `'.\'`’ at the beginning of the filename when referring to it.

The following table shows the supported output formats for exporting from figures and the switch settings to use. In some cases, a format is available both as a MATLAB output filter and as a Ghostscript output filter. All formats except for EMF are supported on both Windows and UNIX platforms.

Note Support for some print options will be removed in a future release. The affected formats have an asterisk (*) next to the option string in the following table. The asterisks provide a link to the Web site which supplies a form for users to give feedback about these changes.

Graphics Format	Bitmap or Vector	Print Command Option String	MATLAB or Ghostscript
BMP monochrome BMP	Bitmap	`-dbmpmono`	Ghostscript
BMP 24-bit BMP	Bitmap	`-dbmp16m`	Ghostscript
BMP 8-bit (256-color) BMP (this format uses a fixed colormap)	Bitmap	`-dbmp256`	Ghostscript
BMP 24-bit	Bitmap	`-dbmp`	MATLAB
EMF	Vector	`-dmeta`	MATLAB
EPS black and white	Vector	`-deps`	MATLAB
EPS color	Vector	`-depsc`	MATLAB
EPS Level 2 black and white	Vector	`-deps2`	MATLAB
EPS Level 2 color	Vector	`-depsc2`	MATLAB
HDF 24-bit	Bitmap	`-dhdf`	MATLAB
ILL (Adobe Illustrator)	Vector	`-dill *`	MATLAB
JPEG 24-bit	Bitmap	`-djpeg`	MATLAB
PBM (plain format) 1-bit	Bitmap	`-dpbm`	Ghostscript

Graphics Format	Bitmap or Vector	Print Command Option String	MATLAB or Ghostscript
PBM (raw format) 1-bit	Bitmap	`-dpbmraw`	Ghostscript
PCX 1-bit	Bitmap	`-dpcxmono`	Ghostscript
PCX 24-bit color PCX file format, three 8-bit planes	Bitmap	`-dpcx24b`	Ghostscript
PCX 8-bit newer color PCX file format (256-color)	Bitmap	`-dpcx256`	Ghostscript
PCX Older color PCX file format (EGA/VGA, 16-color)	Bitmap	`-dpcx16`	Ghostscript
PDF Color PDF file format	Vector	`-dpdf`	Ghostscript
PGM Portable Graymap (plain format)	Bitmap	`-dpgm`	Ghostscript
PGM Portable Graymap (raw format)	Bitmap	`-dpgmraw`	Ghostscript
PNG 24-bit	Bitmap	`-dpng`	MATLAB
PPM Portable Pixmap (plain format)	Bitmap	`-dppm`	Ghostscript
PPM Portable Pixmap (raw format)	Bitmap	`-dppmraw`	Ghostscript

Graphics Format	Bitmap or Vector	Print Command Option String	MATLAB or Ghostscript
SVG Scalable Vector Graphics (For Simulink Models Only)	Vector	`-dsvg`	MATLAB
TIFF 24-bit	Bitmap	`-dtiff` or `-dtiffn`	MATLAB
TIFF preview for EPS files	Bitmap	`-tiff`	

The TIFF image format is supported on all platforms by almost all word processors for importing images. The `-dtiffn` variant writes an uncompressed TIFF. JPEG is a lossy, highly compressed format that is supported on all platforms for image processing and for inclusion into HTML documents on the Web. To create these formats, MATLAB renders the figure using the Z-buffer rendering method and the resulting bitmap is then saved to the specified file.

Printing and Exporting without a Display

On a UNIX platform (including Macintosh), where you can start in MATLAB `nodisplay` mode (`matlab -nodisplay`), you can print using most of the drivers you can use with a display and export to most of the same file formats. The PostScript and Ghostscript devices all function in `nodisplay` mode on UNIX platforms. The graphic devices `-djpeg`, `-dpng`, `-dtiff` (compressed TIFF bitmaps), and `-tiff` (EPS with TIFF preview) work as well, but under `nodisplay` they use Ghostscript to generate output instead of using the drivers built into MATLAB. However, Ghostscript ignores the `-r` option when generating `-djpeg`, `-dpng`, `-dtiff`, and `-tiff` image files. This means that you cannot vary the resolution of image files when running in `nodisplay` mode.

The same is true for the `-noFigureWindows` startup option which suppresses figures on all platforms. On Windows platforms the `-dwin`, `-dwinc`, and `-dsetup` options operate as usual under `-noFigureWindows`. However, the `printpreview` GUI does not function

in this mode. Naturally, the Windows only `-dwin` and `-dwinc` output formats cannot be used on UNIX or Mac platforms with or without a display.

The formats which you cannot generate in `nodisplay` mode on UNIX and Mac platforms are:

- bitmap (`-dbitmap`) — Windows bitmap file
- bmp (`-dbmp...`) — Monochrome and color bitmaps
- hdf (`-dhdf`) — Hierarchical Data Format
- svg (`-dsvg`) — Scalable Vector Graphics file
- tiffn (`-dtiffn`) — TIFF image file, no compression

In addition, uicontrols do not print or export in `nodisplay` mode.

Printing Options

This table summarizes options that you can specify for `print`. The second column links to tutorials that provide operational details. Also see "Resolution Considerations" on page 1-4028 for information on controlling output resolution.

Note Support for some print options will be removed in a future release. The affected formats have an asterisk (*) next to the option string in the following table. The asterisks provide a link to the Web site which supplies a form for users to give feedback about these changes.

Option	Description
`-adobecset` *	PostScript devices only. Use PostScript default character set encoding. See "Early PostScript 1 Printers".
`-append`	PostScript devices only. Append figure to existing PostScript file. See "Settings That Are Driver Specific".

Option	Description
`-cmyk`	PostScript devices only. Print with CMYK colors instead of RGB. See "Setting CMYK Color".
`-ddriver`	Printing only. Printer driver to use. See "Printer Drivers" on page 1-4018 table.
`-dformat`	Exporting only. Graphics format to use. See "Graphics Format Files" table.
`-dsetup` *	Windows printing only. Display the (platform-specific) Print Setup dialog. Settings you make in it are saved, but nothing is printed.
`-fhandle`	Handle of figure to print. Note that you cannot specify both this option and the `-s`*`windowtitle`* option. See "Which Figure Is Printed".
`-loose`	PostScript and Ghostscript printing only. Use loose bounding box for PostScript output. See "Producing Uncropped Figures".
`-noui`	Suppress printing of user interface controls. See "Excluding User Interface Controls".
`-opengl`	Render using the OpenGL algorithm. Note that you cannot specify this method in conjunction with `-zbuffer` or `-painters`. See "Selecting a Renderer".
`-painters`	Render using the Painter's algorithm. Note that you cannot specify this method in conjunction with `-zbuffer` or `-opengl`. See "Selecting a Renderer".
`-P`*`printer`*	Specify name of printer to use. See "Selecting the Printer".

Option	Description
`-rnumber`	PostScript and built-in raster formats, and Ghostscript vector format only. Specify resolution in dots per inch. Defaults to `90` for Simulink, `150` for figures in image formats and when printing in Z-buffer or OpenGL mode, screen resolution for metafiles, and `864` otherwise. Use `-r0` to specify screen resolution. For details, see "Resolution Considerations" on page 1-4028 and "Setting the Resolution".
`-swindowtitle`	Specify name of Simulink system window to print. Note that you cannot specify both this option and the `-fhandle` option. See "Which Figure Is Printed".
`-v`	Windows printing only. Display the Windows Print dialog box. The `v` stands for "verbose mode."
`-zbuffer`	Render using the Z-buffer algorithm. Note that you cannot specify this method in conjunction with `-opengl` or `-painters`. See "Selecting a Renderer".

Resolution Considerations

Use `-rnumber` to specify the resolution of the generated output. In general, using a higher value will yield higher quality output but at the cost of larger output files. It affects the resolution and output size of all MATLAB built-in *raster* formats (which are identified in column four of the table in "Graphics Format Files" on page 1-4022).

Note Built-in graphics formats are generated directly from MATLAB without conversion through the Ghostscript library. Also, in headless (`nodisplay`) mode, writing to certain image formats is not done by built-in drivers, as it is when a display is being used. These formats are `-djpeg`, `-dtiff`, and `-dpng`. Furthermore, the `-dhdf` and `-dbmp` formats cannot be generated in headless mode (but you can substitute `-dbmp16m` for `-dbmp`). See "Printing and Exporting without a Display" on page 1-4025 for details on printing when not using a display.

Unlike the built-in MATLAB formats, graphic output generated via Ghostscript does not directly obey `-r` option settings. However, the intermediate PostScript file generated by MATLAB as input for the Ghostscript processor is affected by the `-r` setting and thus can indirectly influence the quality of the final Ghostscript generated output.

The effect of the `-r` option on output quality can be subtle at ordinary magnification when using the OpenGL or ZBuffer renderers and writing to one of the MATLAB built-in raster formats, or when generating vector output that contains an embedded raster image (for example, PostScript or PDF). The effect of specifying higher resolution is more apparent when viewing the output at higher magnification or when printed, since a larger `-r` setting provides more data to use when scaling the image.

When generating fully vectorized output (as when using the Painters renderer to output a vector format such as PostScript or PDF), the resolution setting affects the degree of detail of the output; setting resolution higher generates crisper output (but small changes in the resolution may have no observable effect). For example, the gap widths of lines that do not use a solid (`'-'`) linestyle can be affected.

Paper Sizes

MATLAB printing supports a number of standard paper sizes. You can select from the following list by setting the `PaperType` property of the figure or selecting a supported paper size from the Print dialog box.

Property Value	Size (Width by Height)
usletter	8.5 by 11 inches
uslegal	8.5 by 14 inches
tabloid	11 by 17 inches
A0	841 by 1189 mm
A1	594 by 841 mm
A2	420 by 594 mm
A3	297 by 420 mm
A4	210 by 297 mm
A5	148 by 210 mm
B0	1029 by 1456 mm
B1	728 by 1028 mm
B2	514 by 728 mm
B3	364 by 514 mm
B4	257 by 364 mm
B5	182 by 257 mm
arch-A	9 by 12 inches
arch-B	12 by 18 inches
arch-C	18 by 24 inches
arch-D	24 by 36 inches
arch-E	36 by 48 inches
A	8.5 by 11 inches
B	11 by 17 inches
C	17 by 22 inches

Property Value	Size (Width by Height)
`D`	22 by 34 inches
`E`	34 by 43 inches

Printing Tips

Setting Default Printer

You can edit the file `printopt.m` to set the default printer type and destination. If you want to set up a new printer, use the operating system printer management utilities. Restart MATLAB if you do not see a printer which is already setup.

Figures with Resize Functions

The `print` command produces a warning when you print a figure having a callback routine defined for the figure `ResizeFcn`. To avoid the warning, set the figure `PaperPositionMode` property to `auto` or select **Auto (Actual Size, Centered)** in the **File > Print Preview** dialog box.

Troubleshooting Microsoft Windows Printing

If you encounter problems such as segmentation violations, general protection faults, or application errors, or the output does not appear as you expect when using Microsoft printer drivers, try the following:

- If your printer is PostScript compatible, print with one of the MATLAB built-in PostScript drivers. There are various PostScript device options that you can use with `print` , which all start with `-dps`.
- The behavior you are experiencing might occur only with certain versions of the print driver. Contact the print driver vendor for information on how to obtain and install a different driver.
- Try printing with one of the MATLAB built-in Ghostscript devices. These devices use Ghostscript to convert PostScript files into other formats, such as HP LaserJet, PCX, Canon BubbleJet, and so on.

- Copy the figure as a Windows Enhanced Metafile using the **Edit > Copy Figure** menu item on the figure window menu or the `print -dmeta` option at the command line. You can then import the file into another application for printing.

  You can set copy options in the figure's **File > Preferences > Copying Options** dialog box. The Windows Enhanced Metafile Clipboard format produces a better quality image than Windows Bitmap.

Printing MATLAB GUIs

You can generally obtain better results when printing a figure window that contains MATLAB `uicontrols` by setting these key properties:

- Set the figure `PaperPositionMode` property to `auto`. This ensures that the printed version is the same size as the on-screen version. With `PaperPositionMode` set to `auto` MATLAB, does not resize the figure to fit the current value of the `PaperPosition`. This is particularly important if you have specified a figure `ResizeFcn`, because if MATLAB resizes the figure during the print operation, `ResizeFcn` is automatically called.

  To set `PaperPositionMode` on the current figure, use the command:

  ```
  set(gcf,'PaperPositionMode','auto')
  ```

- Set the figure `InvertHardcopy` property to `off`. By default, MATLAB changes the figure background color of printed output to white, but does not change the color of `uicontrols`. If you have set the background color, for example, to match the gray of the GUI devices, you must set `InvertHardcopy` to `off` to preserve the color scheme.

  To set `InvertHardcopy` on the current figure, use the command:

  ```
  set(gcf,'InvertHardcopy','off')
  ```

- Use a color device if you want lines and text that are in color on the screen to be written to the output file as colored objects. Black and

white devices convert colored lines and text to black or white to provide the best contrast with the background and to avoid dithering.

- Use the `print` command's `-loose` option to keep a bounding box from being too tightly wrapped around objects contained in the figure. This is important if you have intentionally used space between `uicontrols` or axes and the edge of the figure and you want to maintain this appearance in the printed output.

If you print or export in `nodisplay` mode, none of the `uicontrols` the figure has will be visible. If you run code that adds `uicontrols` to a figure when the figure is invisible, the controls will not print until the figure is made visible.

Printing Interpolated Shading with PostScript Drivers

You can print MATLAB surface objects (such as graphs created with `surf` or `mesh`) using interpolated colors. However, only patch objects that are composed of triangular faces can be printed using interpolated shading.

Printed output is always interpolated in RGB space, not in the colormap colors. This means that if you are using indexed color and interpolated face coloring, the printed output can look different from what is displayed on screen.

PostScript files generated for interpolated shading contain the color information of the graphics object's vertices and require the printer to perform the interpolation calculations. This can take an excessive amount of time and in some cases, printers might time out before finishing the print job. One solution to this problem is to interpolate the data and generate a greater number of faces, which can then be flat shaded.

To ensure that the printed output matches what you see on the screen, print using the `-zbuffer` option. To obtain higher resolution (for example, to make text look better), use the `-r` option to increase the resolution. There is, however, a tradeoff between the resolution and the size of the created PostScript file, which can be quite large at higher resolutions. The default resolution of 150 dpi generally produces good

results. You can reduce the size of the output file by making the figure smaller before printing it and setting the figure `PaperPositionMode` to `auto`, or by just setting the `PaperPosition` property to a smaller size.

Examples

Specifying the Figure to Print

Pass a figure handle as a variable to the function form of `print`. For example:

```
h = figure;
plot(1:4,5:8)
print(h)
```

Save the figure with the handle `h` to a PostScript file named `Figure2`, which can be printed later:

```
print(h,'-dps','Figure2.ps')
```

Pass in a file name as a variable:

```
filename = 'mydata';
print(h, '-dpsc', filename);
```

(Because a file name is specified, the figure will be printed to a file.)

Specifying the Model to Print

Print a noncurrent Simulink model using the `-s` option with the title of the window (in this case, `f14`):

```
print('-sf14')
```

If the window title includes any spaces, you must call the function form rather than the command form of `print`. For example, this command saves the Simulink window title `Thruster Control`:

```
print('-sThruster Control')
```

To print the current system, use:

```
print('-s')
```

For information about issues specific to printing Simulink windows, see the Simulink documentation.

Printing Figures at Screen Size

This example prints a surface plot with interpolated shading. Setting the current figure's (`gcf`) `PaperPositionMode` to `auto` enables you to resize the figure window and print it at the size you see on the screen. See "Printing Options" on page 1-4026 and "Printing Interpolated Shading with PostScript Drivers" on page 1-4033 for information on the `-zbuffer` and `-r200` options.

```
surf(peaks)
shading interp
set(gcf,'PaperPositionMode','auto')
print('-dpsc2','-zbuffer','-r200')
```

For additional details, see "Printing Images" in the MATLAB Graphics documentation.

Batch Processing

You can use the function form of `print` to pass variables containing file names. For example, this `for` loop uses file names stored in a cell array to create a series of graphs and prints each one with a different file name:

```
fnames = {'file1', 'file2', 'file3'};
for k=1:length(fnames)
    surf(peaks)
   print('-dtiff','-r200',fnames{k}) % fnames is a cell of string arra
                                      % each element is a string
```

Tiff Preview

The command

```
print('-depsc','-tiff','-r300','picture1')
```

saves the current figure at 300 dpi, in a color Encapsulated PostScript file named `picture1.eps`. The `-tiff` option creates a 72 dpi TIFF preview, which many word processor applications can display on screen after you import the EPS file. This enables you to view the picture on screen within your word processor and print the document to a PostScript printer using a resolution of 300 dpi.

Alternatives

Select **File > Print** from the figure window to open the Print dialog box and **File > Print Preview** to open the Print Preview GUI. For details, see "How to Print or Export".

See Also

`figure` | `hgsave` | `imwrite` | `orient` | `printdlg` | `printopt` | `saveas`

Purpose

Configure printer defaults

Syntax

```
[pcmd,dev] = printopt
```

Description

`[pcmd,dev] = printopt` returns strings containing the current system-dependent printing command and output device. `printopt` is a file used by `print` to produce the hard-copy output. You can edit the file `printopt.m` to set your default printer type and destination.

`pcmd` and `dev` are platform-dependent strings. `pcmd` contains the command that `print` uses to send a file to the printer. `dev` contains the printer driver or graphics format option for the `print` command. Their defaults are platform dependent.

Platform	Print Command	Driver or Format
Mac and UNIX	`lpr -r`	`-dps2`
Windows	COPY /B %s LPT1:	`-dwin`

See Also

`printdlg` | `print`

printdlg

Purpose

Print dialog box

Syntax

```
printdlg
printdlg(fig)
printdlg('-crossplatform',fig)
printdlg('-setup',fig)
```

Description

`printdlg` prints the current figure.

`printdlg(fig)` creates a modal dialog box from which you can print the figure window identified by the handle `fig`. Note that uimenus do not print.

`printdlg('-crossplatform',fig)` displays the standard cross-platform MATLAB printing dialog rather than the built-in printing dialog box for Microsoft Windows computers. Insert this option before the `fig` argument.

`printdlg('-setup',fig)` forces the printing dialog to appear in a setup mode. If you want to set up a new printer, use the operating system printer management utilities. Restart MATLAB if you don't see the printer which is already setup.

Note It is adviprinter on MATLAB

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

See Also

`print` | `printopt` | `printpreview`

Purpose

Preview figure to print

Contents

"Description" on page 1-4039

"Right Pane Controls" on page 1-4040

"The Layout Tab" on page 1-4041

"The Lines/Text Tab" on page 1-4042

"The Color Tab" on page 1-4044

"The Advanced Tab" on page 1-4046

"Alternatives" on page 1-4047

Syntax

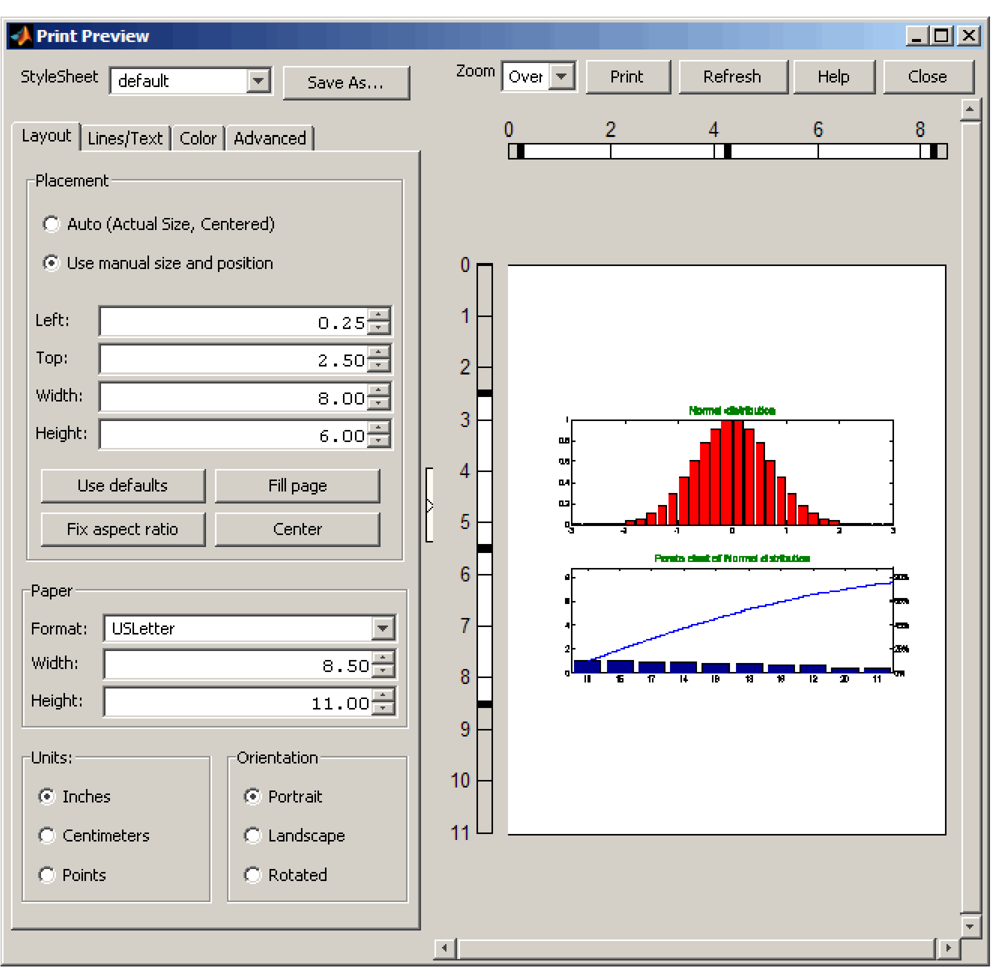
```
printpreview
printpreview(f)
```

Description

`printpreview` displays a dialog box showing the figure in the currently active figure window as it will print. A scaled version of the figure displays in the right-hand pane of the GUI.

`printpreview(f)` displays a dialog box showing the figure having the handle `f` as it will print.

Use the Print Preview dialog box, shown below, to control the layout and appearance of figures before sending them to a printer or print file. Controls are grouped into four tabbed panes: **Layout**, **Lines/Text**, **Color**, and **Advanced**.



Right Pane Controls

You can position and scale plots on the printed page using the rulers in the right-hand pane of the Print Preview dialog. Use the outer ruler handlebars to change margins. Moving them changes plot proportions. Use the center ruler handlebars to change the position of the plot on the page. Plot proportions do not change, but you can move portions of

the plot off the paper. The buttons on that pane let you refresh the plot, close the dialog (preserving all current settings), print the page immediately, or obtain context-sensitive help. Use the **Zoom** box and scroll bars to view and position page elements more precisely.

The Layout Tab

Use the **Layout** tab, shown above, to control the paper format and placement of the plot on printed pages. The following table summarizes the **Layout** options:

Group	Option	Description
Placement	**Auto**	Let MATLAB decide placement of plot on page*
	Use manual...	Specify position parameters for plot on page*
	Top, Left, Width, Height	Standard position parameters in current units
	Use defaults	Revert to default position
	Fill page	Expand figure to fill printable area (see note below)
	Fix aspect ratio	Correct height/width ratio
	Center	Center plot on printed page
Paper	**Format**	U.S. and ISO® sheet size selector
	Width, Height	Sheet size in current units
Units	**Inches**	Use inches as units for dimensions and positions
	Centimeters	Use centimeters as units for dimensions and positions
	Points	Use points as units for dimensions and positions
Orientation	**Portrait**	Upright paper orientation

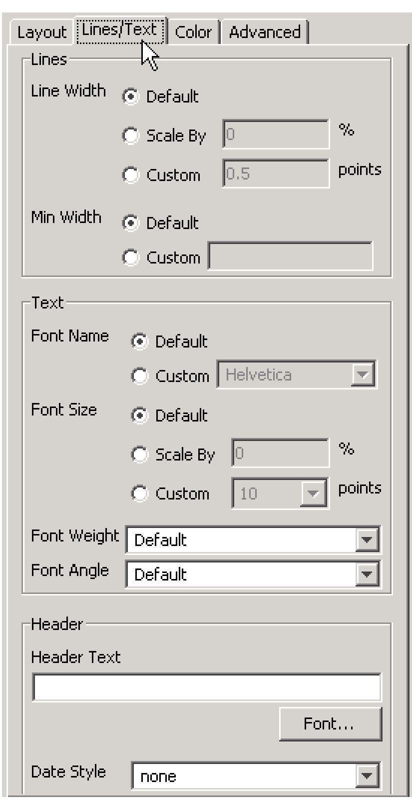
Group	Option	Description
	Landscape	Sideways paper orientation
	Rotated	Currently the same as **Landscape**

* Selecting **Auto** in the Placement group sets the figure `PaperPositionMode` to `'auto'` and disables the controls in that panel. Selecting **Use manual size and position** sets the figure `PaperPositionMode` to `'manual'` and enables the controls. If you set `PaperPositionMode` programmatically, the print preview Placement controls respond accordingly.

Note Selecting the **Fill page** option changes the `PaperPosition` property to fill the page, allowing objects in normalized units to expand to fill the space. If an object within the figure has an absolute size, for example a table, it can overflow the page when objects with normalized units expand. To avoid having objects fall off the page, do not use **Fill page** under such circumstances.

The Lines/Text Tab

Use the **Lines/Text** tab, shown below, to control the line weights, font characteristics, and headers for printed pages. The following table summarizes the **Lines/Text** options:

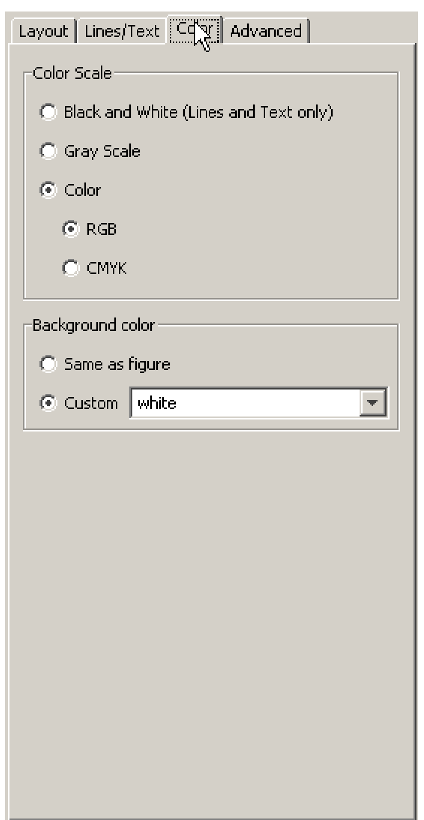


Group	Option	Description
Lines	**Line Width**	Scale all lines by a percentage from 0 upward (100 being no change), print lines at a specified point size, or default line widths used on the plot
	Min Width	Smallest line width (in points) to use when printing; defaults to 0.5 point
Text	**Font Name**	Select a system font for all text on plot, or default to fonts currently used on the plot

Group	Option	Description
	Font Size	Scale all text by a percentage from 0 upward (100 being no change), print text at a specified point size, or default to this
	Font Weight	Select `Normal` ... `Bold` font styling for all text from drop-down menu or default to the font weights used on the plot
	Font Angle	Select `Normal`, `Italic` or `Oblique` font styling for all text from drop-down menu or default to the font angles used on the plot
Header	**Header Text**	Type the text to appear on the header at the upper left of printed pages, or leave blank for no header
	Date Style	Select a date format to have today's date appear at the upper left of printed pages, or `none` for no date

The Color Tab

Use the **Color** tab, shown below, to control how colors are printed for lines and backgrounds. The following table summarizes the **Color** options:

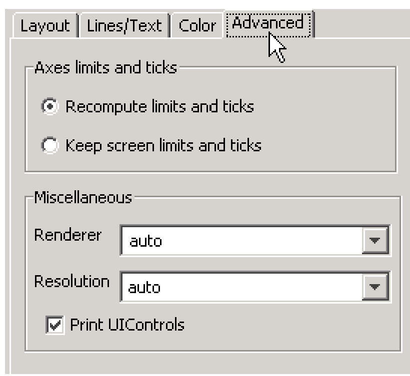


Group	Option	Description
Color Scale	**Black and White**	Select to print lines and text in black and white, but use color for patches and other objects
	Gray Scale	Convert colors to shades of gray on printed pages

Group	Option	Description
	Color	Print everything in color, matching colors on plot; select RGB (default) or CMYK color model for printing
Background Color	**Same as figure**	Print the figure's background color as it is
	Custom	Select a color name, or type a colorspec for the background; white (default) implies no background color, even on colored paper.

The Advanced Tab

Use the **Advanced** tab, shown below, to control finer details of printing, such as limits and ticks, renderer, resolution, and the printing of UIControls. The following table summarizes the **Advanced** options:



Group	Option	Description
Axes limits and ticks	**Recompute limits and ticks**	Redraw x- and y-axes ticks and limits based on printed plot size (default)
	Keep limits and ticks	Use the x- and y-axes ticks and limits shown on the plot when printing the previewed figure
Miscellaneous	**Renderer**	Select a rendering algorithm for printing: `painters`, `zbuffer`, `opengl`, or `auto` (default)
	Resolution	Select resolution to print at in dots per inch: `150`, `300`, `600`, or `auto` (default), or type in any other positive value
	Print UIControls	Print all visible UIControls in the figure (default), or uncheck to exclude them from being printed

Alternatives

Use **File > Print Preview** on the figure window menu to access the Print Preview dialog box, described below. For details, see "Using Print Preview".

See Also

`printdlg` | `pagesetupdlg`

How To

- How to Print or Export

prod

Purpose

Product of array elements

Syntax

```
B = prod(A)
B = prod(A,dim)
B = prod( ___ ,datatype)
```

Description

`B = prod(A)` returns the product of the array elements of `A`.

- If `A` is a vector, then `prod(A)` returns the product of the elements.
- If `A` is a nonempty matrix, then `prod(A)` treats the columns of `A` as vectors and returns a row vector of the products of each column.
- If `A` is an empty 0-by-0 matrix, `prod(A)` returns `1`.
- If `A` is a multidimensional array, then `prod(A)` acts along the first nonsingleton dimension and returns an array of products. The size of this dimension reduces to `1` while the sizes of all other dimensions remain the same.

`prod` computes and returns `B` as `single` if the input, `A`, is `single`. For all other numeric and logical data types, `prod` computes and returns `B` as `double`.

`B = prod(A,dim)` returns the products along dimension `dim`. For example, if `A` is a matrix, `prod(A,2)` is a column vector containing the products of each row.

`B = prod( ___ ,datatype)` multiplies in and returns an array in the class specified by `datatype`, using any of the input arguments in the previous syntaxes. `datatype` can be `'double'` or `'native'`.

Input Arguments

A - Input array

vector | matrix | multidimensional array

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

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

dim - Dimension to operate along

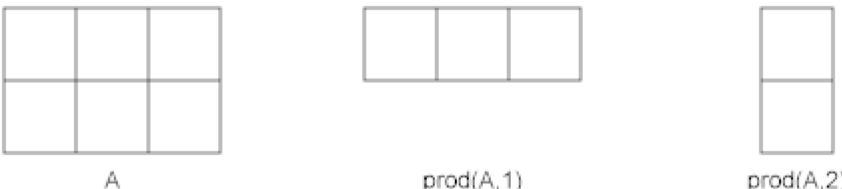
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, the default is the first nonsingleton dimension.

Dimension `dim` indicates the dimension whose length reduces to `1`. The `size(B,dim)` is `1`, while the sizes of all other dimensions remain the same.

Consider a two-dimensional input array, `A`.

- If `dim = 1`, then `prod(A,1)` returns a row vector containing the product of the elements in each column.
- If `dim = 2`, then `prod(A,2)` returns a column vector containing the product of the elements in each row.



`prod` returns `A` if `dim` is greater than `ndims(A)`.

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

datatype - Output class

`'double'` | `'native'`

Output class, specified as `'double'` or `'native'`, defines the data type that the operation is performed in and output in.

- If `datatype` is `'double'`, then `prod` computes and returns a double precision array, regardless of the input data type. For example, if `A` is `single`, then `prod` multiplies in and returns in `double`.
- If `datatype` is `'native'`, `prod` multiplies natively and returns an array with the same data type as the input array `A`. For example, if `A` is an 8-bit unsigned integer, `prod` multiplies natively and returns `B` with data type `uint8`.

Data Types
`char`

Output Arguments

B - Product array

scalar | vector | matrix | multidimensional array

Product array, returned as a scalar, vector, matrix, or multidimensional array.

The class of `B` is as follows:

- If the `datatype` argument is not used and the input is not `single`, then the output is `double`.
- If the `datatype` argument is not used and the input is `single`, then the output is `single`.
- If the `datatype` argument specifies `'double'`, then the output is `double` regardless of the input data type.
- If the `datatype` argument specifies `'native'`, then the output is the same data type as the input.

Definitions

First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to `1`.

For example:

- If `X` is a 1-by-n row vector, then the second dimension is the first nonsingleton dimension of `X`.

- If `X` is a 1-by-0-by-n empty array, then the second dimension is the first nonsingleton dimension of `X`.
- If `X` is a 1-by-1-by-3 array, then the third dimension is the first nonsingleton dimension of `X`.

Examples

Product of Elements in Each Column

Create a 3-by-3 array whose elements correspond to their linear indices.

```
A=[1:3:7;2:3:8;3:3:9]

A =

     1     4     7
     2     5     8
     3     6     9
```

Find the product of the elements in each column.

```
B = prod(A)

B =

     6   120   504
```

The length of the first dimension is `1` , and the length of the second dimension matches `size(A,2)`.

Logical Input with Double Output

Create an array of logical values.

```
A = [true false; true true]

A =

     1     0
     1     1
```

Find the product of the elements in each column.

```
B = prod(A)
```

```
B =

     1     0
```

The output is `double`.

```
class(B)
```

```
ans =

double
```

Product of Elements in Each Row

Create a 3-by-3 array whose elements correspond to their linear indices.

```
A=[1:3:7;2:3:8;3:3:9]
```

```
A =

     1     4     7
     2     5     8
     3     6     9
```

Find the product of the elements in each row and reduce the length of the second dimension to `1`.

```
dim = 2;
B = prod(A,dim)
```

```
B =

    28
```

```
    80
   162
```

The length of the first dimension matches `size(A,1)`, and the length of the second dimension is `1`.

Product of Elements in Each Plane

Create a 3-by-3-by-2 array whose elements correspond to their linear indices.

```
A=[1:3:7;2:3:8;3:3:9];
A(:,:,2)=[10:3:16;11:3:17;12:3:18]
```

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

     1     4     7
     2     5     8
     3     6     9


A(:,:,2) =

    10    13    16
    11    14    17
    12    15    18
```

Find the product of each element in the first plane with its corresponding element in the second plane.

```
dim = 3;
B = prod(A,dim)
```

```
B =

    10    52   112
    22    70   136
    36    90   162
```

The length of the first dimension matches `size(A,1)`, the length of the second dimension matches `size(A,2)`, and the length of the third dimension is `1`.

Single-Precision Input treated as Double

Create a 3-by-3 array of single-precision values.

```
A = single([1200 1500 1800; 1300 1600 1900; 1400 1700 2000])
```

```
A =

        1200            1500            1800
        1300            1600            1900
        1400            1700            2000
```

Find the product of the elements in each row by performing the multiplication in double precision.

```
B = prod(A,2,'double')
```

```
B =

   1.0e+09 *

    3.2400
    3.9520
    4.7600
```

The output is double precision.

```
class(B)
```

```
ans =

double
```

Integer Data Type for Input and Output

Create a 3-by-3 array of 8-bit unsigned integers.

```
A = uint8([1:3:7;2:3:8;3:3:9])
```

```
A =

    1    4    7
    2    5    8
    3    6    9
```

Find the product of the elements in each column natively in `uint8`.

```
B = prod(A,'native')
```

```
B =

    6  120  255
```

The result is an array of 8-bit unsigned integers.

```
class(B)
```

```
ans =

uint8
```

See Also

cumprod | diff | sum | ndims

profile

Purpose

Profile execution time for function

Syntax

```
profile on
profile -history
profile -nohistory
profile -history -historysize integer
profile -timer clock
profile -history -historysize integer -timer clock
profile off
profile resume
profile clear
profile viewer
S = profile('status')
stats = profile('info')
```

Description

The `profile` function helps you debug and optimize MATLAB code files by tracking their execution time. For each MATLAB function, MATLAB local function, or MEX-function in the file, `profile` records information about execution time, number of calls, parent functions, child functions, code line hit count, and code line execution time. Some people use `profile` simply to see the child functions; see also `depfun` for that purpose. To open the Profiler graphical user interface, use the `profile viewer` syntax. By default, Profiler time is CPU time. The total time reported by the Profiler is not the same as the time reported using the `tic` and `toc` functions or the time you would observe using a stopwatch.

Note If your system uses Intel multi-core chips, you may want to restrict the active number of CPUs to 1 for the most accurate and efficient profiling. See “Intel Multi-Core Processors — Setting for Most Accurate Profiling on Windows Systems” or “Intel Multi-Core Processors — Setting for Most Accurate Profiling on Linux Systems” for details on how to do this.

`profile on` starts the Profiler, clearing previously recorded profile statistics. Note the following:

- You can specify all, none, or a subset, of the `history`, `historysize` and `timer` options with the `profile on` syntax.
- You can specify options in any order, including before or after `on`.
- If the Profiler is currently on and you specify `profile` with one of the options, MATLAB software returns an error message and the option has no effect. For example, if you specify `profile timer real`, MATLAB returns the following error: `The profiler has already been started. TIMER cannot be changed.`
- To change options, first specify `profile off`, and then specify `profile on` or `profile resume` with new options.

`profile` **`-history`** records the exact sequence of function calls. The `profile` function records, by default, up to 1,000,000 function entry and exit events. For more than 1,000,000 events, `profile` continues to record other profile statistics, but not the sequence of calls. To change the number of function entry and exit events that the `profile` function records, use the **`historysize`** option. By default, the `history` option is not enabled.

`profile` **`-nohistory`** disables further recording of the history (exact sequence of function calls). Use the `-nohistory` option after having previously set the `-history` option. All other profiling statistics continue to be collected.

`profile` **`-history -historysize`** *`integer`* specifies the number of function entry and exit events to record. By default, `historysize` is set to 1,000,000.

`profile` **`-timer`** *`clock`* specifies the type of time to use. Valid values for *`clock`* are:

- `'cpu'` — The Profiler uses computer time (the default).
- `'real'` — The Profiler uses wall-clock time.

For example, `cpu` time for the `pause` function is typically small, but `real` time accounts for the actual time paused, and therefore would be larger.

`profile` **`-history -historysize`** *`integer`* **`-timer`** *`clock`* specifies all of the options. Any order is acceptable, as is a subset.

`profile` **`off`** stops the Profiler.

`profile` **`resume`** restarts the Profiler without clearing previously recorded statistics.

`profile` **`clear`** clears the statistics recorded by `profile`.

`profile` **`viewer`** stops the Profiler and displays the results in the Profiler window. For more information, see Profiling for Improving Performance in the Desktop Tools and Development Environment documentation.

`S = profile('`**`status`**`')` returns a structure containing information about the current status of the Profiler. The table lists the fields in the order that they appear in the structure.

Field	Values	Default Value
`ProfilerStatus`	`'on'` or `'off'`	off
`DetailLevel`	`'mmex'`	'mmex'
`Timer`	`'cpu'` or `'real'`	'cpu'
`HistoryTracking`	`'on'` or `'off'`	'off'
`HistorySize`	integer	1000000

`stats = profile('`**`info`**`')` displays a structure containing the results. Use this function to access the data generated by `profile`. The table lists the fields in the order that they appear in the structure.

Field	Description
`FunctionTable`	Structure array containing statistics about each function called
`FunctionHistory`	Array containing function call history

Field	Description
`ClockPrecision`	Precision of the `profile` function's time measurement
`ClockSpeed`	Estimated clock speed of the CPU
`Name`	Name of the profiler

The `FunctionTable` field is an array of structures, where each structure contains information about one of the functions or local functions called during execution. The following table lists these fields in the order that they appear in the structure.

Field	Description
`CompleteName`	Full path to `FunctionName`, including local functions
`FunctionName`	Function name; includes local functions
`FileName`	Full path to `FunctionName`, with file extension, excluding local functions
`Type`	MATLAB functions, MEX-functions, and many other types of functions including MATLAB local functions, nested functions, and anonymous functions
`NumCalls`	Number of times the function was called
`TotalTime`	Total time spent in the function and its child functions
`TotalRecursiveTime`	No longer used.
`Children`	`FunctionTable` indices to child functions
`Parents`	`FunctionTable` indices to parent functions

Field	Description
ExecutedLines	Array containing line-by-line details for the function being profiled. Column 1: Number of the line that executed. If a line was not executed, it does not appear in this matrix. Column 2: Number of times the line was executed Column 3: Total time spent on that line. Note: The sum of Column 3 entries does not necessarily add up to the function's TotalTime.
IsRecursive	BOOLEAN value: Logical 1 (true) if recursive, otherwise logical 0 (false)
PartialData	BOOLEAN value: Logical 1 (true) if function was modified during profiling, for example by being edited or cleared. In that event, data was collected only up until the point when the function was modified.

Examples

Profile, View Results, and Save Profile Data as HTML

This example profiles the MATLAB magic command and then displays the results in the Profiler window. The example then retrieves the profile data on which the HTML display is based and uses the profsave command to save the profile data in HTML form.

```
profile on
plot(magic(35))
profile viewer
p = profile('info');
profsave(p,'profile_results')
```

Profile, Save Profile Data to a MAT-File, and View Results

Another way to save profile data is to store it in a MAT-file. This example stores the profile data in a MAT-file, clears the profile data from memory, and then loads the profile data from the MAT-file. This example also shows a way to bring the reloaded profile data into the Profiler graphical interface as live profile data, not as a static HTML page.

```
p = profile('info');
save myprofiledata p
clear p
load myprofiledata
profview(0,p)
```

Profile and Show Results Including History

This example illustrates an effective way to view the results of profiling when the `history` option is enabled. The history data describes the sequence of functions entered and exited during execution. The `profile` command returns history data in the `FunctionHistory` field of the structure it returns. The history data is a 2-by-`n` array. The first row contains Boolean values, where `0` means entrance into a function and `1` means exit from a function. The second row identifies the function being entered or exited by its index in the `FunctionTable` field. This example reads the history data and displays it in the MATLAB Command Window.

```
profile on -history
plot(magic(4));
p = profile('info');

for n = 1:size(p.FunctionHistory,2)
 if p.FunctionHistory(1,n)==0
        str = 'entering function: ';
 else
        str = 'exiting function: ';
 end
 disp([str p.FunctionTable(p.FunctionHistory(2,n)).FunctionName])
```

```
end
```

See Also

`depdir` | `depfun` | `mlint` | `profsave`

How To

- Profiling for Improving Performance
- “Profiling Parallel Code”

Purpose

Save profile report in HTML format

Syntax

```
profsave
profsave(profinfo)
profsave(profinfo,dirname)
```

Description

`profsave` executes the `profile('info')` function and saves the results in HTML format. `profsave` creates a separate HTML file for each function listed in the `FunctionTable` field of the structure returned by `profile`. By default, `profsave` stores the HTML files in a subfolder of the current folder named `profile_results`.

`profsave(profinfo)` saves the profiling results, `profinfo`, in HTML format. `profinfo` is a structure of profiling information returned by the `profile('info')` function.

`profsave(profinfo,dirname)` saves the profiling results, `profinfo`, in HTML format. `profsave` creates a separate HTML file for each function listed in the `FunctionTable` field of `profinfo` and stores them in the folder specified by `dirname`.

Examples

Run `profile` and save the results.

```
profile on
plot(magic(5))
profile off
profsave(profile('info'),'myprofile_results')
```

See Also

`profile`

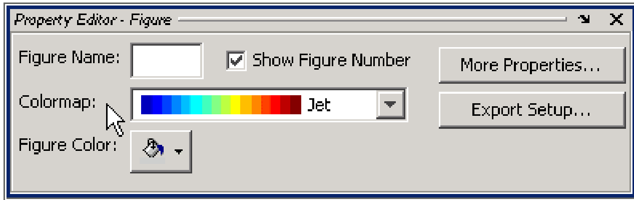
How To

- Profiling for Improving Performance

propedit

Purpose

Open Property Editor



Syntax

```
propedit
propedit(handle_list)
```

Description

`propedit` starts the Property Editor, a graphical user interface to the properties of graphics objects. If no current figure exists, `propedit` will create one.

`propedit(handle_list)` edits the properties for the object (or objects) in `handle_list`.

Starting the Property Editor enables plot editing mode for the figure.

See Also

`inspect` | `plotedit` | `propertyeditor`

Purpose

Open built-in property page for control

Syntax

```
h.propedit
propedit(h)
```

Description

`h.propedit` requests the control to display its built-in property page. Note that some controls do not have a built-in property page. For those controls, this command fails.

`propedit(h)` is an alternate syntax for the same operation.

Tips

COM functions are available on Microsoft Windows systems only.

See Also

`inspect` | `get (COM)`

properties

Purpose

Class property names

Syntax

```
properties('classname')
properties(obj)
p = properties(...)
```

Description

`properties('classname')` displays the names of the public properties for the MATLAB class named by `classname`. The `properties` function also displays inherited properties.

`properties(obj)` `obj` can be either a scalar object or an array of objects. When `obj` is scalar, `properties` also returns dynamic properties. See "Dynamic Properties — Adding Properties to an Instance" for information on using dynamic properties.

`p = properties(...)` returns the property names in a cell array of strings.

Definitions

A property is public when its `GetAccess` attribute value is `public` and its `Hidden` attribute value is `false` (default values for these attributes). See "Property Attributes" for a complete list of attributes.

`properties` is also a MATLAB class-definition keyword. See `classdef` for more information on class definition keywords.

Examples

Retrieve the names of the public properties of class `memmapfile` and store the result in a cell array of strings:

```
p = properties('memmapfile');
p
ans =

    'writable'
    'offset'
    'format'
    'repeat'
    'filename'
```

Construct an instance of the MException class and get its properties names:

```
me = MException('Msg:ID','MsgText');
properties(me)
Properties for class MException:

    identifier
    message
    cause
    stack
```

See Also

fieldnames | events | methods

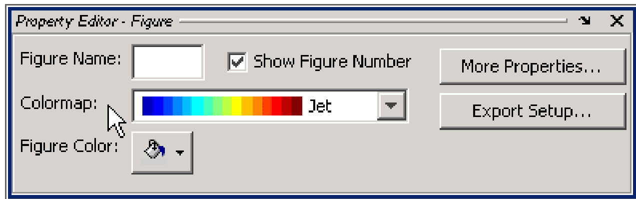
Tutorials

- "Properties"

propertyeditor

Purpose

Show or hide **Property Editor**



Syntax

```
propertyeditor('on')
propertyeditor('off')
propertyeditor
propertyeditor(figure_handle,...)
```

Description

`propertyeditor('on')` displays the Property Editor tool on the current figure.

`propertyeditor('off')` hides the Property Editor on the current figure.

`propertyeditor` toggles the visibility of the Property Editor on the current figure. You can also use `propertyeditor('toggle')`instead for the same functionality.

`propertyeditor(figure_handle,...)` displays or hides the Property Editor on the figure specified by `figure_handle`.

Tips

If you call `propertyeditor` in a MATLAB program and subsequent lines depend on the Property Editor being fully initialized, follow it by `drawnow` to ensure complete initialization.

Alternatives

To collectively enable **Plotting Tools**, use the large Plotting Tool icon on the figure toolbar. To collectively disable the **Plotting Tools**, use the smaller icon . Open or close the **Property Editor** tool from the figure's **View** menu. For details, see "The Property Editor".

See Also

`plottools` | `plotbrowser` | `figurepalette` | `inspect`

psi

Purpose

Psi (polygamma) function

Note The syntax `Y = psi(k0:k1,X)` has been removed.

Syntax

```
Y = psi(X)
Y = psi(k,X)
Y = psi(k0:k1,X)
```

Description

`Y = psi(X)` evaluates the ψ function for each element of array `X`. `X` must be real and nonnegative. The ψ function, also known as the digamma function, is the logarithmic derivative of the gamma function

$$\begin{aligned}\psi(x) &= \text{digamma}(x)\\ &= \frac{d(\log(\Gamma(x)))}{dx}\\ &= \frac{d(\Gamma(x))/dx}{\Gamma(x)}\end{aligned}$$

`Y = psi(k,X)` evaluates the `k`th derivative of ψ at the elements of `X`. `psi(0,X)` is the digamma function, `psi(1,X)` is the trigamma function, `psi(2,X)` is the tetragamma function, etc.

`Y = psi(k0:k1,X)` evaluates derivatives of order `k0` through `k1` at `X`. `Y(k,j)` is the `(k-1+k0)`th derivative of ψ, evaluated at `X(j)`.

Examples

Example 1

Use the `psi` function to calculate Euler's constant, γ.

```
format long
-psi(1)
ans =
   0.57721566490153

-psi(0,1)
```

```
ans =
   0.57721566490153
```

Example 2

The trigamma function of 2, `psi(1,2)`, is the same as $(\pi^2/6) - 1$.

```
format long
psi(1,2)
ans =
   0.64493406684823

pi^2/6 - 1
ans =
   0.64493406684823
```

Example 3

This code produces the first page of Table 6.1 in Abramowitz and Stegun [1].

```
x = (1:.005:1.250)';
[x gamma(x) gammaln(x) psi(0:1,x)' x-1]
```

Example 4

This code produces a portion of Table 6.2 in [1].

```
psi(2:3,1:.01:2)'
```

References

[1] Abramowitz, M. and I. A. Stegun, *Handbook of Mathematical Functions*, Dover Publications, 1965, Sections 6.3 and 6.4.

See Also

`gamma` | `gammainc` | `gammaln`

publish

Purpose

Generate view of MATLAB file in specified format

Syntax

```
publish(file)
publish(file,format)

publish(file,Name,Value)
publish(file,options)

my_doc = publish(file,___)
```

Description

`publish(file)` generates a view of a MATLAB file in HTML format for sharing your code. For example, `publish('myfile.m')` executes the code in `myfile.m` using the base workspace and saves the formatted code and results in `/html/myfile.html`. The `html` subfolder is relative to the `file` folder.

`publish(file,format)` generates a view of a MATLAB file in the specified file format. The resulting files save to the `html` subfolder for all file formats.

`publish(file,Name,Value)` generates a view of the MATLAB file, `file`, with options specified by one or more name-value pair arguments.

`publish(file,options)` uses the `options` structure to customize the output, which is useful when you want to preconfigure and save your options for repeated use. The `options` structure fields and values correspond to names and values of name-value pair arguments, respectively.

`my_doc = publish(file,___)` generates a view of the MATLAB file `file`, and returns the path of the resulting output file as `my_doc`.

Input Arguments

file - MATLAB file

string

Full or partial path of the MATLAB file for which you want to generate a presentation view, specified as a string.

Example: `'myfile.m'`

format - Output format of published file

`'html'` (default) | `'doc'` | `'latex'` | `'ppt'` | `'xml'` | `'pdf'`

Output format of published MATLAB file, specified as one of the following string values.

Output Format	String Value
Hypertext Markup Language	`'html'` (default)
Microsoft Word	`'doc'`
LaTeX	`'latex'`
Microsoft PowerPoint	`'ppt'`
Extensible Markup Language	`'xml'`
Portable Document Format	`'pdf'`

Example: `publish('myfile.m','ppt');`

options - Options for published output

MATLAB structure

Options for published output, specified as a structure. Use the options structure instead of name-value pair arguments when you want to reuse the same configuration for publishing multiple MATLAB code files.

The `options` structure field and values correspond to names and values of the name-value pair arguments, respectively.

For example, to specify the PDF output format and the output folder `C:\myPublishedOutput`, use:

```
options = struct('format','pdf','outputDir','C:\myPublishedOutput')
```

Name-Value Pair Arguments

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

Example: `'format','latex','showCode',false` specifies LaTeX output file format and excludes the code from the output.

Output Options

'format' - Published output file format

`'html'` (default) | `'doc'` | `'latex'` | `'ppt'` | `'xml'` | `'pdf'`

Published output file format, specified as the comma-separated pair consisting of `'format'` and one of the following string values.

Output Format	String Value
Hypertext Markup Language	`'html'` (default)
Microsoft Word	`'doc'`
LaTeX	`'latex'`
Microsoft PowerPoint	`'ppt'`
Extensible Markup Language	`'xml'`
Portable Document Format	`'pdf'`

'outputDir' - Output folder

`''` (default) | full path

Output folder to which the published document is saved, specified as the comma-separated pair consisting of `'outputDir'` and the full path. You must specify the full path as a string, for example `'C:\myPublishedOutput'`.

The default value, `''`, specifies the `html` subfolder of the current folder.

'stylesheet' - Extensible Stylesheet language (XSL) file

`''` (default) | full path to XSL file name

Extensible Stylesheet Language (XSL) file to use when publishing MATLAB code to HTML, XML, or LaTeX format, specified as the comma-separated pair consisting of `'stylesheet'` and the full path to the XSL file. The full path must be a string, for example, `'C:\myStylesheet\stylesheet.xsl'`

The default value, `''`, specifies the MATLAB default style sheet.

Figure Options

'createThumbnail' - Thumbnail image creation

`true` (default) | `false`

Thumbnail image creation for the published document, specified as the comma-separated pair consisting of `'createThumbnail'` and either `true` or `false`. You can use this thumbnail to represent your file in HTML pages.

'figureSnapMethod' - Published figure window appearance

`'entireGUIWindow'` (default) | `'print'` | `'getframe'` | `'entireFigureWindow'`

Appearance of published figure windows, including window decorations and background color, specified as the comma-separated pair consisting of `'figureSnapMethod'` and one of the following string values.

String Value	Window Decorations		Background Color	
	GUIs	Figures	GUIs	Figures
`'entireGUIWindow'` (default)	Included	Excluded	Matches screen	White
`'print'`	Excluded	Excluded	White	White

(Continued)

'getframe'	Excluded	Excluded	Matches screen	Matches screen
'entireFigureWindow'	Included	Included	Matches screen	Matches screen

'imageFormat' - Published image file format

'png' | 'epsc2' | 'jpg'

Published image file format, specified as the comma-separated pair consisting of 'imageFormat' and one of the following string values.

'format' Value	Valid 'imageFormat' Value	Default 'imageFormat' Value
'doc'	Any image format that your installed version of Microsoft Office can import, including 'png' , 'jpg', 'bmp', and 'tiff'. If the 'figureSnapMethod' is 'print', then you can also specify 'eps', 'epsc', 'eps2', 'ill', 'meta', and 'pdf'.	'png'
'html'	Any format publishes successfully. Ensure that the tools you use to view and process the output files can display the output format you specify.	'png'

(Continued)

'latex'	Any format publishes successfully. Ensure that the tools you use to view and process the output files can display the output format you specify.	'epsc2', except when 'figureSnapMethod' is any one of the following: • 'getframe' • 'entireFigureWindow' • 'entireGUIWindow' and the snapped window is a GUI window In these cases the default is 'png'
'pdf'	'bmp' or 'jpg'.	'bmp'
'ppt'	Any format that your installed version of Microsoft Office can import, including 'png', 'jpg', 'bmp', and 'tiff'.	'png'
'xml'	Any format publishes successfully. Ensure that the tools you use to view and process the output files can display the output format you specify.	'png'

'maxHeight' - Maximum height of published images
[ ] (default) | positive integer value

Maximum height of published images that the code generates, specified as the comma-separated pair consisting of `'maxHeight'` and one of the following values:

- `[]` (default)—Unrestricted height. This value is always used when the `'format'` value is `'pdf'`.
- Positive integer value that specifies the image height in pixels.

'maxWidth' - Maximum width of published images
`''` (default) | positive integer value

Maximum width of an image that the code generates, specified as the comma-separated pair consisting of `'maxWidth'` and one of the following values:

- `[]` (default)—Unrestricted width. This value is always used when the `'format'` value is `'pdf'`.
- Positive integer value that specifies the image width in pixels.

'useNewFigure' - Create new figure
`true` (default) | `false`

A logical value that determines if MATLAB creates a new Figure window for figures that the code generates, specified as the comma-separated pair consisting of `'useNewFigure'` and one of the following:

- `true` (default)—If the code generates a figure, then MATLAB creates a Figure window with a white background, and at the default size before publishing.
- `false`—MATLAB does not create a figure window.

  This value enables you to use a figure with different properties for publishing. For example, open a Figure window, change the size and background color, and then publish your code. Figures in your published document use the characteristics of the figure you opened before publishing.

Code Options

'evalCode' - Option to run code

`true` (default) | `false`

Option to evaluate code and include the MATLAB output in the published view, specified as a logical value.

'catchError' - Error handling during publishing

`true` (default) | `false`

Error handling during publishing, specified as the comma-separated pair consisting of `'catchError'` and one of the following logical values:

- `true` (default)—MATLAB continues publishing and includes the error in the published file.
- `false`—MATLAB displays the error at the command line and does not produce a published file.

'codeToEvaluate' - Additional code to evaluate during publishing

string

Additional code to evaluate during publishing, specified as the comma-separated pair consisting of `'codeToEvaluate'` and the string with the corresponding code. Use this option to specify code that does not appear in the MATLAB file, for example to set the value of an input argument for a function being published.

If this option is unspecified, MATLAB evaluates only the code in the MATLAB file you are publishing.

'maxOutputLines' - Maximum number of lines

`Inf` (default) | Nonnegative integer value

Maximum number of lines in MATLAB output per cell evaluated during publishing, specified as the comma-separated pair consisting of `'maxOutputLines'` and one of the following values:

- `Inf` (default)—MATLAB includes all output lines in the published output.

- Nonnegative integer—MATLAB truncates the number of lines in the output at the number of lines you specify.

'showCode' - Option to include code in published file

`true` (default) | `false`

Option to include code in published file, specified as the comma-separated pair consisting of `'maxOutputLines'` and a logical value.

Output Arguments

my_doc - Path of published output file

Path of published output file, returned as a string.

Tips

- If `'format` is `'html'`, MATLAB includes the code being published at the end of the published HTML file as comments, even when you set the `'showCode'` option to `false`. Because MATLAB includes the code as comments, this code does not display in a Web browser. This enables you to use the `grabcode` function to extract the MATLAB code from an HTML file, even when the file does not display the code.
- MATLAB does not preserve syntax highlighting when you set `'format'` to any of the following:
  - `'latex'`
  - `'doc'`
  - `'ppt'`

Definitions

Window Decorations

Window decorations include window title bar, toolbar, menu bar, and window border.

Syntax Highlighting

Syntax highlighting colors various elements in code to help you identify these elements while reading or editing code. By default, keywords are blue, strings are purple, and unterminated strings are maroon.

Examples

Generate HTML View of MATLAB Script

Generate an HTML view of the code, MATLAB results, and comments in a MATLAB script.

Copy the example file, `fourier_demo2.m`, to your current folder.

```
copyfile(fullfile(matlabroot,'help','techdoc',...
'matlab_env','examples','fourier_demo2.m'),'.','f')
```

Generate an HTML view of the MATLAB file.

```
 publish('fourier_demo2.m');
```

The `publish` command executes the code for each cell in `fourier_demo2.m`, and saves the file to `/html/fourier_demo2.html`.

View the HTML file.

```
web('html/fourier_demo2.html')
```

Generate View of MATLAB Script in Microsoft Word Format

Generate a Microsoft Word view of the code, MATLAB results, and comments in a MATLAB script.

Copy the example file to your current folder.

```
copyfile(fullfile(matlabroot,'help','techdoc',...
'matlab_env','examples','fourier_demo2.m'),'.','f')
```

Publish the file in Microsoft Word format.

```
publish('fourier_demo2.m','doc');
```

View the published output.

```
winopen('html/fourier_demo2.doc')
```

Publish MATLAB Script Using Name-Value Pairs to Customize the Output

Generate a Microsoft Word view of the code, MATLAB results, and comments in a MATLAB script. Use name-value pair arguments to include window decorations in the published output.

Copy the example file to your current folder.

```
copyfile(fullfile(matlabroot,'help','techdoc',...
'matlab_env','examples','fourier_demo2.m'),'.','f')
```

Publish the example MATLAB file to Microsoft Word format.

```
publish('fourier_demo2.m', 'figureSnapMethod', 'entireFigureWindow')
```

The `'figureSnapMethod','entireFigureWindow'` name-value pair argument specifies to include the window decorations in the figures, and to match the figure background color to the screen color.

View the published output.

```
web('html/fourier_demo2.html')
```

Customize `publish` Output Using Options Structure

Generate a Microsoft Word view of the code, MATLAB results, and comments in a MATLAB script. Use a structure to customize the published output.

Specifying options as a structure is useful when you want to preconfigure and save your options for repeated use.

Copy the example file to your current folder.

```
copyfile(fullfile(matlabroot,'help','techdoc',...
'matlab_env','examples','fourier_demo2.m'),'.','f')
```

Define options to customize the published output as a structure, `options_doc_nocode`.

```
options_doc_nocode.format = 'doc';
options_doc_nocode.showCode = false;
```

Publish the file, specifying the options structure.

```
publish('fourier_demo2.m',options_doc_nocode);
```

Save File Path of Published Script to Variable

Generate an HTML view of a MATLAB script, and save the path of the published HTML file to a variable.

This example assumes that the current folder is `C:\my_MATLAB_files`.

Copy the example file to your current folder.

```
copyfile(fullfile(matlabroot,'help','techdoc',...
'matlab_env','examples','fourier_demo2.m'),'.','f')
```

Publish the MATLAB file, and save the path of the resulting published HTML file to an output variable.

```
mydoc = publish('fourier_demo2.m')
```

```
mydoc =

C:\my_MATLAB_files\html\fourier_demo2.html
```

See Also

grabcode | notebook

Concepts

- “Publishing MATLAB Code”
- “Publishing Markup”

PutCharArray

Purpose

Store character array in Automation server

Syntax

MATLAB Client

```
h.PutCharArray('varname', 'workspace', 'string')
PutCharArray(h, 'varname', 'workspace', 'string')
invoke(h, 'PutCharArray', 'varname', 'workspace', 'string')
```

IDL Method Signature

```
PutCharArray([in] BSTR varname, [in] BSTR workspace,
[in] BSTR string)
```

Microsoft Visual Basic Client

```
PutCharArray(varname As String, workspace As String,
string As String)
```

Description

`PutCharArray` stores the character array in `string` in the specified `workspace` of the server attached to handle `h`, assigning to it the variable `varname`. The `workspace` argument can be either `base` or `global`.

Tips

The character array specified in the `string` argument can have any dimensions. However, `PutCharArray` changes the dimensions to a 1–by-n column-wise representation, where n is the number of characters in the array. Executing the following commands in MATLAB illustrates this behavior:

```
h = actxserver('matlab.application');
chArr = ['abc'; 'def'; 'ghk']
chArr =
abc
def
ghk

h.PutCharArray('Foo', 'base', chArr)
tstArr = h.GetCharArray('Foo', 'base')
tstArr =
adgbehcfk
```

Server function names, like PutCharArray, are case sensitive when using the dot notation syntax shown in the Syntax section.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

Examples

Store string str in the base workspace of the server using PutCharArray.

MATLAB Client

```
h = actxserver('matlab.application');
h.PutCharArray('str', 'base', ...
   'He jests at scars that never felt a wound.')

S = h.GetCharArray('str', 'base')
S =
    He jests at scars that never felt a wound.
```

Visual Basic .NET Client

This example uses the Visual Basic MsgBox command to control flow between MATLAB and the Visual Basic Client.

```
Dim Matlab As Object
Try
    Matlab = GetObject(, "matlab.application")
Catch e As Exception
    Matlab = CreateObject("matlab.application")
End Try
MsgBox("MATLAB window created; now open it...")
```

Open the MATLAB window, then click **Ok**.

```
Matlab.PutCharArray("str", "base", _
       "He jests at scars that never felt a wound.")
MsgBox("In MATLAB, type" & vbCrLf _
  & "str")
```

In the MATLAB window type `str`; MATLAB displays:

```
str =
He jests at scars that never felt a wound.
```

Click **Ok**.

```
MsgBox("closing MATLAB window...")
```

Click **Ok** to close and terminate MATLAB.

```
Matlab.Quit()
```

See Also

`GetCharArray` | `PutWorkspaceData` | `GetWorkspaceData` | `Execute`

Purpose

Matrix in Automation server workspace

Syntax

MATLAB Client

```
h.PutFullMatrix('varname','workspace',xreal,ximag)
PutFullMatrix(h,'varname','workspace',xreal,ximag)
```

IDL Method Signature

```
PutFullMatrix([in] BSTR varname, [in] BSTR workspace,
[in] SAFEARRAY(double) xreal, [in] SAFEARRAY(double) ximag)
```

Microsoft Visual Basic Client

```
PutFullMatrix([in] varname As String, [in] workspace As String,
[in] xreal As Double, [in] ximag As Double)
```

Description

`h.PutFullMatrix('varname','`*`workspace`*`',xreal,ximag)` stores a matrix in the specified `workspace` of the server attached to handle `h` and assigns it to variable `varname`. Use `xreal` and `ximag` for the real and imaginary parts of the matrix. The matrix cannot be a scalar, an empty array, or have more than two dimensions. The values for *`workspace`* are `base` or `global`.

`PutFullMatrix(h,'varname','`*`workspace`*`',xreal,ximag)` is an alternate syntax.

For VBScript clients, use the `GetWorkspaceData` and `PutWorkspaceData` functions to pass numeric data to and from the MATLAB workspace. These functions use the `variant` data type instead of `safearray` which is not supported by VBScript.

Examples

This example uses a MATLAB client to write a matrix to the base workspace of the server:

```
h = actxserver('matlab.application');
h.PutFullMatrix('M','base',rand(5),zeros(5))
%Use one output for real values only
xreal = h.GetFullMatrix('M','base',zeros(5),zeros(5))
```

This example uses a Visual Basic .NET client to write a matrix to the base workspace of the server:

```
Dim MatLab As Object
Dim XReal(4, 4) As Double
Dim XImag(4, 4) As Double
Dim ZReal(4, 4) As Double
Dim ZImag(4, 4) As Double
Dim i, j As Integer

For i = 0 To 4
   For j = 0 To 4
   XReal(i, j) = Rnd() * 6
   XImag(i, j) = 0
   Next j
Next i

Matlab = CreateObject("matlab.application")
MatLab.PutFullMatrix("M","base",XReal,XImag)
MatLab.GetFullMatrix("M","base",ZReal,ZImag)
```

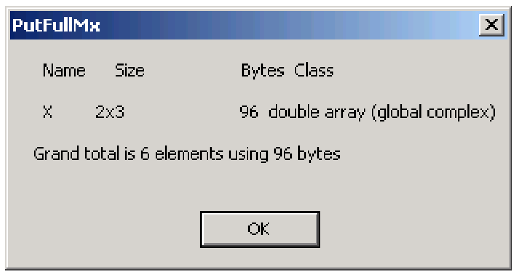
Use a MATLAB client to write a matrix to the global workspace of the server:

```
h = actxserver('matlab.application');
h.PutFullMatrix('X','global',[1 3 5; 2 4 6],...
  [1 1 1; 1 1 1])
h.invoke('Execute','whos global')
```

Use a Visual Basic .NET client to write a matrix to the global workspace of the server:

```
Dim MatLab As Object
```

```
Dim XReal(1,2) As Double
Dim XImag(1,2) As Double
Dim result As String
Dim i,j As Integer

For i = 0 To 1
 For j = 0 To 2
  XReal(i,j) = (j * 2 + 1) + i
  XImag(i,j) = 1
 Next j
Next i

Matlab = CreateObject("matlab.application")
MatLab.PutFullMatrix("X","global",XReal,XImag)
result = Matlab.Execute("whos global")
MsgBox(result)
```



See Also

GetFullMatrix | PutWorkspaceData | Execute

How To

- “MATLAB COM Automation Server Interface”
- “Exchanging Data with the Server”

PutWorkspaceData

Purpose

Data in Automation server workspace

Syntax

MATLAB Client

```
h.PutWorkspaceData('varname','workspace',data)
PutWorkspaceData(h,'varname','workspace',data)
```

IDL Method Signature

```
PutWorkspaceData([in] BSTR varname, [in] BSTR workspace,
[in] VARIANT data)
```

Microsoft Visual Basic Client

```
PutWorkspaceData(varname As String, workspace As String,
data As Object)
```

Description

`h.PutWorkspaceData('varname','`*`workspace`*`',data)` stores `data` in the `workspace` of the server attached to handle `h` and assigns it to `varname`. The values for *`workspace`* are `base` or `global`.

`PutWorkspaceData(h,'varname','`*`workspace`*`',data)` is an alternate syntax.

Use `PutWorkspaceData` to pass numeric and character array data respectively to the server. Do *not* use `PutWorkspaceData` on sparse arrays, structures, or function handles. Use the `Execute` method for these data types.

The `GetWorkspaceData` and `PutWorkspaceData` functions pass numeric data as a `variant` data type. These functions are especially useful for VBScript clients as VBScript does not support the `safearray` data type used by `GetFullMatrix` and `PutFullMatrix`.

Examples

Create an array in a MATLAB client and put it in the base workspace of the MATLAB Automation server:

```
h = actxserver('matlab.application');
for i = 0:6
   data(i+1)  = i * 15;
end
```

```
h.PutWorkspaceData('A','base',data)
```

Create an array in a Visual Basic .NET client and put it in the base workspace of the MATLAB Automation server:

1. Create the Visual Basic application. Use the `MsgBox` command to control flow between MATLAB and the application:

   ```
   Dim Matlab As Object
   Dim data(6) As Double
   Dim i As Integer
   MatLab = CreateObject("matlab.application")
   For i = 0 To 6
      data(i)  = i * 15
   Next i
   MatLab.PutWorkspaceData("A","base",data)
   MsgBox("In MATLAB, type" & vbCrLf & "A")
   ```

2. Open the MATLAB window and type `A`. MATLAB displays:

   ```
   A =
        0    15    30    45    60    75    90
   ```

3. Click **Ok** to close and terminate MATLAB.

See Also

`GetWorkspaceData` | `PutFullMatrix` | `PutCharArray` | `Execute`

How To

- "Executing Commands in the MATLAB Server"
- "Exchanging Data with the Server"

pwd

Purpose

Identify current folder

Syntax

```
pwd
currentFolder = pwd
```

Description

`pwd` displays the MATLAB current folder.

`currentFolder = pwd` returns the current folder as a string to `currentFolder`.

Alternatives

- Use the Current Folder toolbar.

See Also

`cd` | `dir`

Purpose

Quasi-minimal residual method

Syntax

```
x = qmr(A,b)
qmr(A,b,tol)
qmr(A,b,tol,maxit)
qmr(A,b,tol,maxit,M)
qmr(A,b,tol,maxit,M1,M2)
qmr(A,b,tol,maxit,M1,M2,x0)
[x,flag] = qmr(A,b,...)
[x,flag,relres] = qmr(A,b,...)
[x,flag,relres,iter] = qmr(A,b,...)
[x,flag,relres,iter,resvec] = qmr(A,b,...)
```

Description

`x = qmr(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and should be large and sparse. The column vector `b` must have length `n`. You can specify `A` as a function handle, `afun`, such that `afun(x,'notransp')` returns `A*x` and `afun(x,'transp')` returns `A'*x`.

"Parameterizing Functions" explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `qmr` converges, a message to that effect is displayed. If `qmr` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`qmr(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `qmr` uses the default, `1e-6`.

`qmr(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `qmr` uses the default, `min(n,20)`.

`qmr(A,b,tol,maxit,M)` and `qmr(A,b,tol,maxit,M1,M2)` use preconditioners `M` or `M = M1*M2` and effectively solve the system `inv(M)*A*x = inv(M)*b` for `x`. If `M` is `[]` then `qmr` applies no

preconditioner. `M` can be a function handle `mfun` such that `mfun(x,'notransp')` returns `M\x` and `mfun(x,'transp')` returns `M'\x`.

`qmr(A,b,tol,maxit,M1,M2,x0)` specifies the initial guess. If `x0` is `[]`, then `qmr` uses the default, an all zero vector.

`[x,flag] = qmr(A,b,...)` also returns a convergence flag.

Flag	Convergence
0	`qmr` converged to the desired tolerance `tol` within `maxit` iterations.
1	`qmr` iterated `maxit` times but did not converge.
2	Preconditioner `M` was ill-conditioned.
3	The method stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `qmr` became too small or too large to continue computing.

Whenever `flag` is not `0`, the solution `x` returned is that with minimal norm residual computed over all the iterations. No messages are displayed if the `flag` output is specified.

`[x,flag,relres] = qmr(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is `0`, `relres <= tol`.

`[x,flag,relres,iter] = qmr(A,b,...)` also returns the iteration number at which `x` was computed, where `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = qmr(A,b,...)` also returns a vector of the residual norms at each iteration, including `norm(b-A*x0)`.

Examples

Using qmr with a Matrix Input

This example shows how to use `qmr` with a matrix input. The code:

```
n = 100;
on = ones(n,1);
```

```
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8; maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x = qmr(A,b,tol,maxit,M1,M2);
```

displays the message:

```
qmr converged at iteration 9 to a solution...
with relative residual
5.6e-009
```

Using qmr with a Function Handle

This example replaces the matrix `A` in the previous example with a handle to a matrix-vector product function `afun`. The example is contained in a file `run_qmr` that

- Calls `qmr` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_qmr` are available to `afun`.

The following shows the code for `run_qmr`:

```
function x1 = run_qmr
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x1 = qmr(@afun,b,tol,maxit,M1,M2);

    function y = afun(x,transp_flag)
       if strcmp(transp_flag,'transp')      % y = A'*x
          y = 4 * x;
```

```
            y(1:n-1) = y(1:n-1) - 2 * x(2:n);
            y(2:n) = y(2:n) - x(1:n-1);
         elseif strcmp(transp_flag,'notransp') % y = A*x
            y = 4 * x;
            y(2:n) = y(2:n) - 2 * x(1:n-1);
            y(1:n-1) = y(1:n-1) - x(2:n);
         end
      end
   end
```

When you enter

```
x1=run_qmr;
```

MATLAB software displays the message

```
qmr converged at iteration 9 to a solution with relative residual
5.6e-009
```

Using a Preconditioner

This example demonstrates the use of a preconditioner.

1

Load `A = west0479`, a real 479-by-479 nonsymmetric sparse matrix:

```
load west0479;
A = west0479;
```

2

Define `b` so that the true solution is a vector of all ones:

```
b = full(sum(A,2));
```

3

Set the tolerance and maximum number of iterations:

```
tol = 1e-12; maxit = 20;
```

4

Use qmr to find a solution at the requested tolerance and number of iterations:

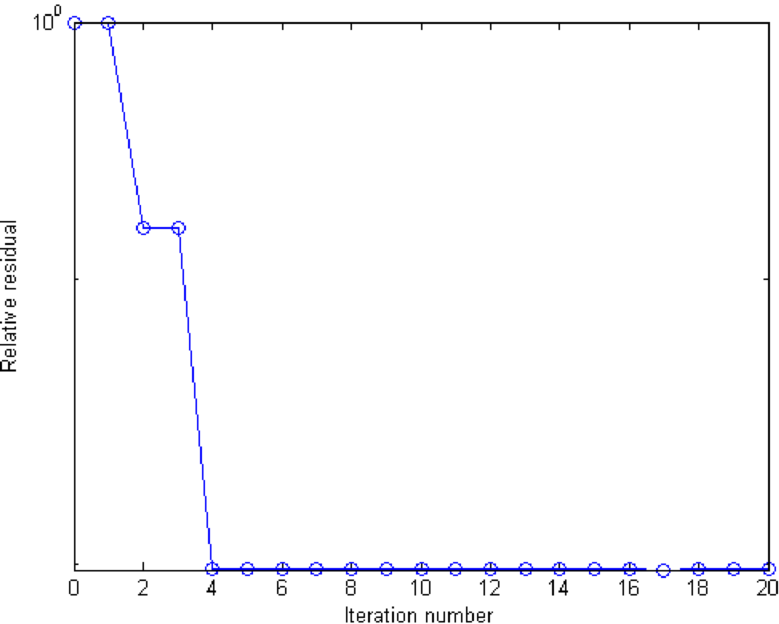
```
[x0,fl0,rr0,it0,rv0] = qmr(A,b,tol,maxit);
```

fl0 is 1 because qmr does not converge to the requested tolerance 1e-12 within the requested 20 iterations. The seventeenth iterate is the best approximate solution and is the one returned as indicated by it0 = 17. MATLAB stores the residual history in rv0.

5

Plot the behavior of qmr:

```
semilogy(0:maxit,rv0/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



The plot shows that the solution does not converge. You can use a preconditioner to improve the outcome.

6

Create the preconditioner with `ilu`, since the matrix `A` is nonsymmetric:

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));
Error using ilu
There is a pivot equal to zero. Consider decreasing
the drop tolerance or consider using the 'udiag' option.
```

MATLABcannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

7

You can try again with a reduced drop tolerance, as indicated by the error message:

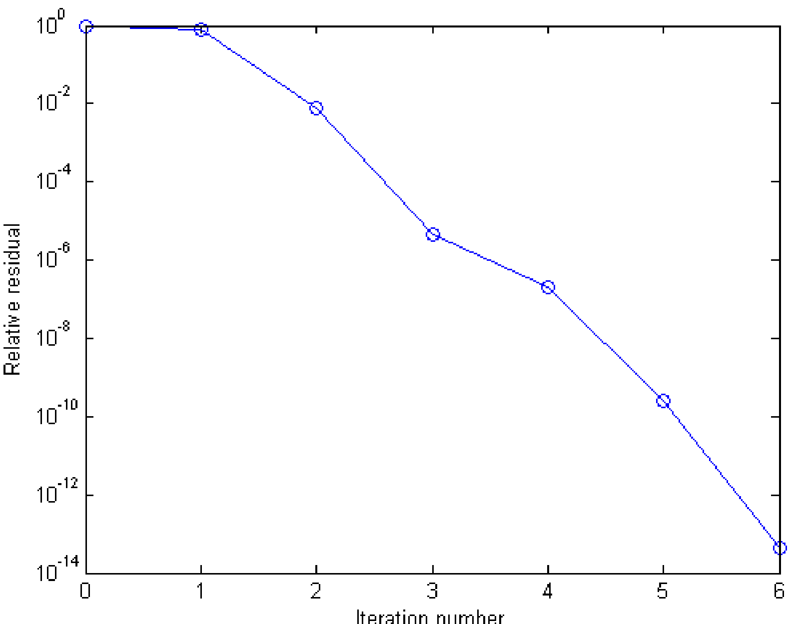
```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = qmr(A,b,tol,maxit,L,U);
```

`fl1` is 0 because `qmr` drives the relative residual to `4.1410e-014` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the sixth iteration (the value of `it1`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output `rv1(1)` is `norm(b)`, and the output `rv1(7)` is `norm(b-A*x2)`.

8

You can follow the progress of `qmr` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0):

```
semilogy(0:it1,rv1/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Freund, Roland W. and Nöel M. Nachtigal, “QMR: A quasi-minimal residual method for non-Hermitian linear systems,” *SIAM Journal: Numer. Math.* 60, 1991, pp. 315–339.

See Also

`bicg` | `bicgstab` | `cgs` | `gmres` | `lsqr` | `luinc` | `minres` | `pcg` | `symmlq` | `function_handle` | `mldivide`

Purpose

Orthogonal-triangular decomposition

Syntax

```
[Q,R] = qr(A)
[Q,R] = qr(A,0)
[Q,R,E] = qr(A)
[Q,R,E] = qr(A,'matrix')
[Q,R,e] = qr(A,'vector')
[Q,R,e] = qr(A,0)
X = qr(A)
X = qr(A,0)
R = qr(A)
R = qr(A,0)
[C,R] = qr(A,B)
[C,R,E] = qr(A,B)
[C,R,E] = qr(A,B,'matrix')
[C,R,e] = qr(A,B,'vector')
[C,R] = qr(A,B,0)
[C,R,e] = qr(A,B,0)
```

Description

`[Q,R] = qr(A)`, where `A` is `m`-by-`n`, produces an `m`-by-`n` upper triangular matrix `R` and an `m`-by-`m` unitary matrix `Q` so that `A = Q*R`.

`[Q,R] = qr(A,0)` produces the economy-size decomposition. If `m > n`, only the first `n` columns of `Q` and the first `n` rows of `R` are computed. If `m<=n`, this is the same as `[Q,R] = qr(A)`.

If A is full:

`[Q,R,E] = qr(A)` or `[Q,R,E] = qr(A,'matrix')` produces unitary `Q`, upper triangular `R` and a permutation matrix `E` so that `A*E = Q*R`. The column permutation `E` is chosen so that `abs(diag(R))` is decreasing.

`[Q,R,e] = qr(A,'vector')` returns the permutation information as a vector instead of a matrix. That is, `e` is a row vector such that `A(:,e) = Q*R`.

`[Q,R,e] = qr(A,0)` produces an economy-size decomposition in which `e` is a permutation vector, so that `A(:,e) = Q*R`.

`X = qr(A)` and `X = qr(A,0)` return a matrix `X` such that `triu(X)` is the upper triangular factor `R`.

If A is sparse:

`R = qr(A)` computes a `Q`-less `QR` decomposition and returns the upper triangular factor `R`. Note that `R = chol(A'*A)`. Since `Q` is often nearly full, this is preferred to `[Q,R] = QR(A)`.

`R = qr(A,0)` produces economy-size `R`. If `m>n`, `R` has only `n` rows. If `m<=n`, this is the same as `R = qr(A)`.

`[Q,R,E] = qr(A)` or `[Q,R,E] = qr(A,'matrix')` produces unitary `Q`, upper triangular `R` and a permutation matrix `E` so that `A*E = Q*R`. The column permutation `E` is chosen to reduce fill-in in R.

`[Q,R,e] = qr(A,'vector')` returns the permutation information as a vector instead of a matrix. That is, `e` is a row vector such that `A(:,e) = Q*R`.

`[Q,R,e] = qr(A,0)` produces an economy-size decomposition in which `e` is a permutation vector, so that `A(:,e) = Q*R`.

`[C,R] = qr(A,B)`, where `B` has as many rows as `A`, returns `C = Q'*B`. The least-squares solution to `A*X = B` is `X = R\C`.

`[C,R,E] = qr(A,B)` or `[C,R,E] = qr(A,B,'matrix')`, also returns a fill-reducing ordering. The least-squares solution to `A*X = B` is `X = E*(R\C)`.

`[C,R,e] = qr(A,B,'vector')` returns the permutation information as a vector instead of a matrix. That is, the least-squares solution to `A*X = B` is `X(e,:) = R\C`.

`[C,R] = qr(A,B,0)` produces economy-size results. If `m>n`, `C` and `R` have only `n` rows. If `m<=n`, this is the same as `[C,R] = qr(A,B)`.

`[C,R,e] = qr(A,B,0)` additionally produces a fill-reducing permutation vector `e`. In this case, the least-squares solution to `A*X = B` is `X(e,:) = R\C`.

Examples

Find the least squares approximate solution to `A*x = b` with the `Q`-less `QR` decomposition and one step of iterative refinement:

```
if issparse(A), R = qr(A);
else R = triu(qr(A)); end
x = R\(R'\(A'*b));
r = b - A*x;
err = R\(R'\(A'*r));
x = x + err;
```

See Also

`lu` | `chol` | `null` | `orth` | `qrdelete` | `qrinsert` | `qrupdate`

qrdelete

Purpose

Remove column or row from QR factorization

Syntax

```
[Q1,R1] = qrdelete(Q,R,j)
[Q1,R1] = qrdelete(Q,R,j,'col')
[Q1,R1] = qrdelete(Q,R,j,'row')
```

Description

`[Q1,R1] = qrdelete(Q,R,j)` returns the QR factorization of the matrix `A1`, where `A1` is `A` with the column `A(:,j)` removed and `[Q,R] = qr(A)` is the QR factorization of `A`.

`[Q1,R1] = qrdelete(Q,R,j,'col')` is the same as `qrdelete(Q,R,j)`.

`[Q1,R1] = qrdelete(Q,R,j,'row')` returns the QR factorization of the matrix `A1`, where `A1` is `A` with the row `A(j,:)` removed and `[Q,R] = qr(A)` is the QR factorization of `A`.

Examples

```
A = magic(5);
[Q,R] = qr(A);
j = 3;
[Q1,R1] = qrdelete(Q,R,j,'row');

Q1 =
    0.5274   -0.5197   -0.6697   -0.0578
    0.7135    0.6911    0.0158    0.1142
    0.3102   -0.1982    0.4675   -0.8037
    0.3413   -0.4616    0.5768    0.5811

R1 =
   32.2335   26.0908   19.9482   21.4063   23.3297
         0  -19.7045  -10.9891    0.4318   -1.4873
         0         0   22.7444    5.8357   -3.1977
         0         0         0  -14.5784    3.7796
```

returns a valid QR factorization, although possibly different from

```
A2 = A;
A2(j,:) = [];
[Q2,R2] = qr(A2)
```

```
Q2 =
   -0.5274    0.5197    0.6697   -0.0578
   -0.7135   -0.6911   -0.0158    0.1142
   -0.3102    0.1982   -0.4675   -0.8037
   -0.3413    0.4616   -0.5768    0.5811

R2 =
  -32.2335  -26.0908  -19.9482  -21.4063  -23.3297
         0   19.7045   10.9891   -0.4318    1.4873
         0         0  -22.7444   -5.8357    3.1977
         0         0         0  -14.5784    3.7796
```

Algorithms

The `qrdelete` function uses a series of Givens rotations to zero out the appropriate elements of the factorization.

See Also

`planerot` | `qr` | `qrinsert`

qrinsert

Purpose

Insert column or row into QR factorization

Syntax

```
[Q1,R1] = qrinsert(Q,R,j,x)
[Q1,R1] = qrinsert(Q,R,j,x,'col')
[Q1,R1] = qrinsert(Q,R,j,x,'row')
```

Description

`[Q1,R1] = qrinsert(Q,R,j,x)` returns the QR factorization of the matrix `A1`, where `A1` is `A = Q*R` with the column `x` inserted before `A(:,j)`. If `A` has `n` columns and `j = n+1`, then `x` is inserted after the last column of `A`.

`[Q1,R1] = qrinsert(Q,R,j,x,'col')` is the same as `qrinsert(Q,R,j,x)`.

`[Q1,R1] = qrinsert(Q,R,j,x,'row')` returns the QR factorization of the matrix `A1`, where `A1` is `A = Q*R` with an extra row, `x`, inserted before `A(j,:)`.

Examples

```
A = magic(5);
[Q,R] = qr(A);
j = 3;
x = 1:5;
[Q1,R1] = qrinsert(Q,R,j,x,'row')

Q1 =
    0.5231    0.5039   -0.6750    0.1205    0.0411    0.0225
    0.7078   -0.6966    0.0190   -0.0788    0.0833   -0.0150
    0.0308    0.0592    0.0656    0.1169    0.1527   -0.9769
    0.1231    0.1363    0.3542    0.6222    0.6398    0.2104
    0.3077    0.1902    0.4100    0.4161   -0.7264   -0.0150
    0.3385    0.4500    0.4961   -0.6366    0.1761    0.0225

R1 =
   32.4962   26.6801   21.4795   23.8182   26.0031
         0   19.9292   12.4403    2.1340    4.3271
         0         0   24.4514   11.8132    3.9931
         0         0         0   20.2382   10.3392
```

```
         0         0         0         0   16.1948
         0         0         0         0         0
```

returns a valid QR factorization, although possibly different from

```
A2 = [A(1:j-1,:); x; A(j:end,:)];
[Q2,R2] = qr(A2)

Q2 =
   -0.5231    0.5039    0.6750   -0.1205    0.0411    0.0225
   -0.7078   -0.6966   -0.0190    0.0788    0.0833   -0.0150
   -0.0308    0.0592   -0.0656   -0.1169    0.1527   -0.9769
   -0.1231    0.1363   -0.3542   -0.6222    0.6398    0.2104
   -0.3077    0.1902   -0.4100   -0.4161   -0.7264   -0.0150
   -0.3385    0.4500   -0.4961    0.6366    0.1761    0.0225

R2 =
  -32.4962  -26.6801  -21.4795  -23.8182  -26.0031
         0   19.9292   12.4403    2.1340    4.3271
         0         0  -24.4514  -11.8132   -3.9931
         0         0         0  -20.2382  -10.3392
         0         0         0         0   16.1948
         0         0         0         0         0
```

Algorithms

The `qrinsert` function inserts the values of `x` into the `j`th column (row) of `R`. It then uses a series of Givens rotations to zero out the nonzero elements of `R` on and below the diagonal in the `j`th column (row).

See Also

`planerot` | `qr` | `qrdelete`

qrupdate

Purpose

Rank 1 update to QR factorization

Syntax

```
[Q1,R1] = qrupdate(Q,R,u,v)
```

Description

`[Q1,R1] = qrupdate(Q,R,u,v)` when `[Q,R] = qr(A)` is the original QR factorization of `A`, returns the QR factorization of `A + u*v'`, where `u` and `v` are column vectors of appropriate lengths.

Tips

`qrupdate` works only for full matrices.

Examples

The matrix

```
mu = sqrt(eps)

mu =

   1.4901e-08

A = [ones(1,4); mu*eye(4)];
```

is a well-known example in least squares that indicates the dangers of forming `A'*A`. Instead, we work with the QR factorization – orthonormal Q and upper triangular R.

```
 [Q,R] = qr(A);
```

As we expect, `R` is upper triangular.

```
R =

   -1.0000   -1.0000   -1.0000   -1.0000
         0    0.0000    0.0000    0.0000
         0         0    0.0000    0.0000
         0         0         0    0.0000
         0         0         0         0
```

In this case, the upper triangular entries of `R`, excluding the first row, are on the order of `sqrt(eps)`.

Consider the update vectors

```
 u = [-1 0 0 0 0]'; v = ones(4,1);
```

Instead of computing the rather trivial QR factorization of this rank one update to `A` from scratch with

```
[QT,RT] = qr(A + u*v')

QT =

     0     0     0     0     1
    -1     0     0     0     0
     0    -1     0     0     0
     0     0    -1     0     0
     0     0     0    -1     0

RT =

   1.0e-007 *

   -0.1490         0         0         0
         0   -0.1490         0         0
         0         0   -0.1490         0
         0         0         0   -0.1490
         0         0         0         0
```

we may use `qrupdate`.

```
[Q1,R1] = qrupdate(Q,R,u,v)

Q1 =

   -0.0000   -0.0000   -0.0000   -0.0000    1.0000
    1.0000   -0.0000   -0.0000   -0.0000    0.0000
```

```
    0.0000    1.0000   -0.0000   -0.0000    0.0000
    0.0000    0.0000    1.0000   -0.0000    0.0000
   -0.0000   -0.0000   -0.0000    1.0000    0.0000

R1 =

   1.0e-007 *
    0.1490    0.0000    0.0000    0.0000
         0    0.1490    0.0000    0.0000
         0         0    0.1490    0.0000
         0         0         0    0.1490
         0         0         0         0
```

Note that both factorizations are correct, even though they are different.

Algorithms

`qrupdate` uses the algorithm in section 12.5.1 of the third edition of *Matrix Computations* by Golub and van Loan. `qrupdate` is useful since, if we take `N = max(m,n)`, then computing the new QR factorization from scratch is roughly an $O(N^3)$ algorithm, while simply updating the existing factors in this way is an $O(N^2)$ algorithm.

References

[1] Golub, Gene H. and Charles Van Loan, *Matrix Computations*, Third Edition, Johns Hopkins University Press, Baltimore, 1996

See Also

`cholupdate` | `qr`

Purpose

Numerically evaluate integral, adaptive Simpson quadrature

Compatibility

`quad` will be removed in a future release. Use `integral` instead.

Syntax

```
q = quad(fun,a,b)
q = quad(fun,a,b,tol)
q = quad(fun,a,b,tol,trace)
[q,fcnt] = quad(...)
```

Description

Quadrature is a numerical method used to find the area under the graph of a function, that is, to compute a definite integral.

$$q = \int_a^b f(x)dx$$

`q = quad(fun,a,b)` tries to approximate the integral of function `fun` from `a` to `b` to within an error of `1e-6` using recursive adaptive Simpson quadrature. `fun` is a function handle. Limits `a` and `b` must be finite. The function `y = fun(x)` should accept a vector argument `x` and return a vector result `y`, the integrand evaluated at each element of `x`.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`q = quad(fun,a,b,tol)` uses an absolute error tolerance `tol` instead of the default which is `1.0e-6`. Larger values of `tol` result in fewer function evaluations and faster computation, but less accurate results. In MATLAB version 5.3 and earlier, the `quad` function used a less reliable algorithm and a default relative tolerance of `1.0e-3`.

`q = quad(fun,a,b,tol,trace)` with non-zero `trace` shows the values of `[fcnt a b-a Q]` during the recursion.

`[q,fcnt] = quad(...)` returns the number of function evaluations.

The function `quadl` may be more efficient with high accuracies and smooth integrands.

The list below contains information to help you determine which quadrature function in MATLAB to use:

- The quad function may be most efficient for low accuracies with nonsmooth integrands.
- The quadl function may be more efficient than quad at higher accuracies with smooth integrands.
- The quadgk function may be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The quadv function vectorizes quad for an array-valued fun.
- If the interval is infinite, [a,Inf), then for the integral of fun(x) to exist, fun(x) must decay as x approaches infinity, and quadgk requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but quadgk can be used if fun(x) decays fast enough.
- The quadgk function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint c like log|x-c| or $|x-c|^p$ for p >= -1/2. If the function is singular at points inside (a,b), write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with quadgk, and add the results.

Examples

To compute the integral

$$\int_0^2 \frac{1}{x^3 - 2x - 5} dx,$$

write a function myfun that computes the integrand:

```
function y = myfun(x)
y = 1./(x.^3-2*x-5);
```

Then pass `@myfun`, a function handle to `myfun`, to `quad`, along with the limits of integration, `0` to `2`:

```
Q = quad(@myfun,0,2)

Q =

   -0.4605
```

Alternatively, you can pass the integrand to `quad` as an anonymous function handle `F`:

```
F = @(x)1./(x.^3-2*x-5);
Q = quad(F,0,2);
```

Algorithms

`quad` implements a low order method using an adaptive recursive Simpson's rule.

Diagnostics

`quad` may issue one of the following warnings:

`'Minimum step size reached'` indicates that the recursive interval subdivision has produced a subinterval whose length is on the order of roundoff error in the length of the original interval. A nonintegrable singularity is possible.

`'Maximum function count exceeded'` indicates that the integrand has been evaluated more than 10,000 times. A nonintegrable singularity is likely.

`'Infinite or Not-a-Number function value encountered'` indicates a floating point overflow or division by zero during the evaluation of the integrand in the interior of the interval.

References

[1] Gander, W. and W. Gautschi, "Adaptive Quadrature – Revisited," BIT, Vol. 40, 2000, pp. 84-101. This document is also available at `http://www.inf.ethz.ch/personal/gander`.

See Also

`quad2d` | `dblquad` | `quadgk` | `quadl` | `quadv` | `trapz` | `triplequad` | `function_handle` | `integral` | `integral2` | `integral3`

How To

- "Anonymous Functions"

Purpose

Numerically evaluate double integral, tiled method

Syntax

```
q = quad2d(fun,a,b,c,d)
[q,errbnd] = quad2d(...)
q = quad2d(fun,a,b,c,d,param1,val1,param2,val2,...)
```

Description

`q = quad2d(fun,a,b,c,d)` approximates the integral of `fun(x,y)` over the planar region $a \le x \le b$ and $c(x) \le y \le d(x)$. `fun` is a function handle, `c` and `d` may each be a scalar or a function handle.

All input functions must be vectorized. The function `Z=fun(X,Y)` must accept 2-D matrices `X` and `Y` of the same size and return a matrix `Z` of corresponding values. The functions `ymin=c(X)` and `ymax=d(X)` must accept matrices and return matrices of the same size with corresponding values.

`[q,errbnd] = quad2d(...)`. `errbnd` is an approximate upper bound on the absolute error, `|Q - I|`, where `I` denotes the exact value of the integral.

`q = quad2d(fun,a,b,c,d,param1,val1,param2,val2,...)` performs the integration as above with specified values of optional parameters:

`AbsTol`	absolute error tolerance
`RelTol`	relative error tolerance

`quad2d` attempts to satisfy `ERRBND <= max(AbsTol,RelTol*|Q|)`. This is absolute error control when `|Q|` is sufficiently small and relative error control when `|Q|` is larger. A default tolerance value is used when a tolerance is not specified. The default value of `AbsTol` is 1e-5. The default value of `RelTol` is `100*eps(class(Q))`. This is also the minimum value of `RelTol`. Smaller `RelTol` values are automatically increased to the default value.

`MaxFunEvals`	Maximum allowed number of evaluations of `fun` reached.

The MaxFunEvals parameter limits the number of vectorized calls to fun. The default is 2000.

FailurePlot	Generate a plot if MaxFunEvals is reached.

Setting FailurePlot to true generates a graphical representation of the regions needing further refinement when MaxFunEvals is reached. No plot is generated if the integration succeeds before reaching MaxFunEvals. These (generally) 4-sided regions are mapped to rectangles internally. Clusters of small regions indicate the areas of difficulty. The default is false.

Singular	Problem may have boundary singularities

With Singular set to true, quad2d will employ transformations to weaken boundary singularities for better performance. The default is true. Setting Singular to false will turn these transformations off, which may provide a performance benefit on some smooth problems.

Examples

Example 1

Integrate $y\sin(x)+x\cos(y)$ over $\pi \le x \le 2\pi$, $0 \le y \le \pi$. The true value of the integral is $-\pi^2$.

```
Q = quad2d(@(x,y) y.*sin(x)+x.*cos(y),pi,2*pi,0,pi)
```

Example 2

Integrate $[(x+y)^{1/2}(1+x+y)^2]^{-1}$ over the triangle $0 \le x \le 1$ and $0 \le y \le 1-x$. The integrand is infinite at (0,0). The true value of the integral is $\pi/4-1/2$.

```
fun = @(x,y) 1./(sqrt(x + y) .* (1 + x + y).^2 )
```

In Cartesian coordinates:

```
ymax = @(x) 1 - x;
```

```
Q = quad2d(fun,0,1,0,ymax)
```

In polar coordinates:

```
polarfun = @(theta,r) fun(r.*cos(theta),r.*sin(theta)).*r;
rmax = @(theta) 1./(sin(theta) + cos(theta));
Q = quad2d(polarfun,0,pi/2,0,rmax)
```

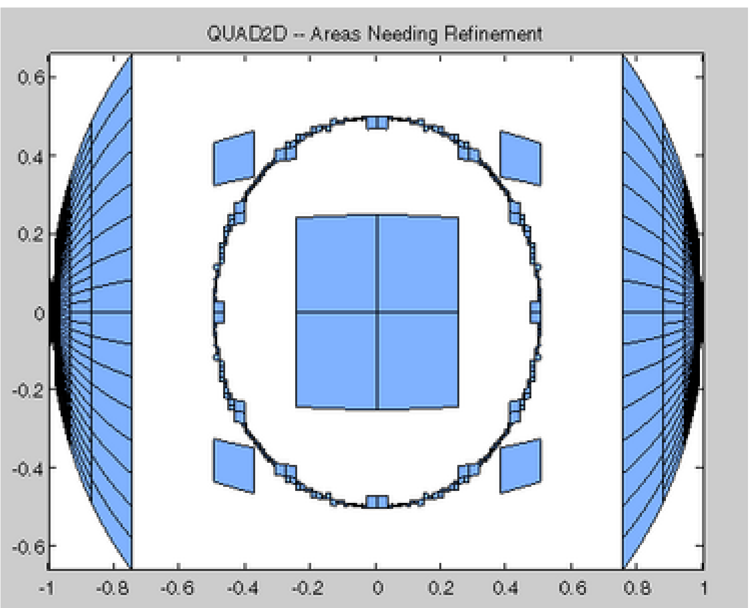
Limitations

`quad2d` begins by mapping the region of integration to a rectangle. Consequently, it may have trouble integrating over a region that does not have four sides or has a side that cannot be mapped smoothly to a straight line. If the integration is unsuccessful, some helpful tactics are leaving `Singular` set to its default value of `true`, changing between Cartesian and polar coordinates, or breaking the region of integration into pieces and adding the results of integration over the pieces.

For example:

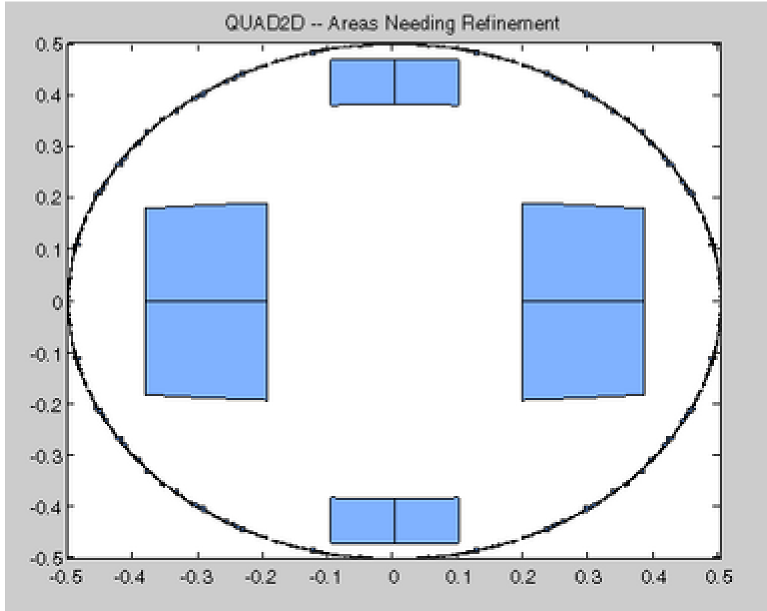
```
fun = @(x,y)abs(x.^2 + y.^2 - 0.25);
c = @(x)-sqrt(1 - x.^2);
d = @(x)sqrt(1 - x.^2);
quad2d(fun,-1,1,c,d,'AbsTol',1e-8,...
    'FailurePlot',true,'Singular',false)
Warning: Reached the maximum number of function ...
     evaluations (2000). The result fails the  ...
     global error test.
```

The failure plot shows two areas of difficulty, near the points `(-1,0)` and `(1,0)` and near the circle $x^2 + y^2 = 0.25$:



Changing the value of `Singular` to `true` will cope with the geometric singularities at `(-1,0)` and `(1,0)`. The larger shaded areas may need refinement but are probably not areas of difficulty.

```
Q = quad2d(fun,-1,1,c,d,'AbsTol',1e-8, ...
     'FailurePlot',true,'Singular',true)
Warning: Reached the maximum number of function ...
     evaluations (2000). The result passes the ...
     global error test.
```

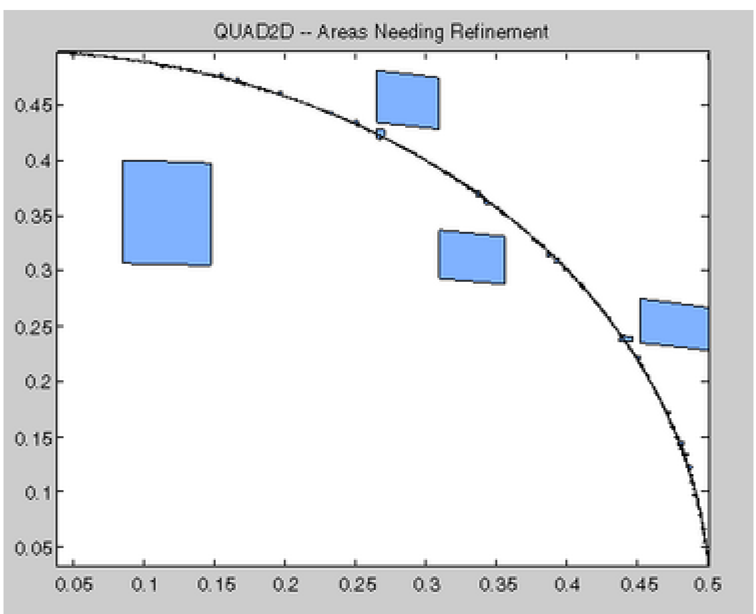


From here you can take advantage of symmetry:

```
Q = 4*quad2d(fun,0,1,0,d,'Abstol',1e-8,...
     'Singular',true, 'FailurePlot',true)
```

However, the code is still working very hard near the singularity. It may not be able to provide higher accuracy:

```
Q = 4*quad2d(fun,0,1,0,d,'Abstol',1e-10,...
     'Singular',true,'FailurePlot',true)
Warning: Reached the maximum number of function ...
     evaluations (2000). The result passes the ...
     global error test.
```



At higher accuracy, a change in coordinates may work better.

```
polarfun = @(theta,r) fun(r.*cos(theta),r.*sin(theta)).*r;
Q = 4*quad2d(polarfun,0,pi/2,0,1,'AbsTol',1e-10)
```

It is best to put the singularity on the boundary by splitting the region of integration into two parts:

```
Q1 = 4*quad2d(polarfun,0,pi/2,0,0.5,'AbsTol',5e-11);
Q2 = 4*quad2d(polarfun,0,pi/2,0.5,1,'AbsTol',5e-11);
Q = Q1 + Q2
```

References

[1] L.F. Shampine, "Matlab Program for Quadrature in 2D." *Applied Mathematics and Computation.* Vol. 202, Issue 1, 2008, pp. 266–274.

See Also

`dblquad` | `quad` | `quadl` | `quadv` | `quadgk` | `triplequad` | `function_handle` | `integral` | `integral2` | `integral3`

How To

- "Anonymous Functions"

quadgk

Purpose

Numerically evaluate integral, adaptive Gauss-Kronrod quadrature

Syntax

```
q = quadgk(fun,a,b)
[q,errbnd] = quadgk(fun,a,b)
[q,errbnd] = quadgk(fun,a,b,param1,val1,param2,val2,...)
```

Description

`q = quadgk(fun,a,b)` attempts to approximate the integral of a scalar-valued function `fun` from `a` to `b` using high-order global adaptive quadrature and default error tolerances. The function `y = fun(x)` should accept a vector argument `x` and return a vector result `y`, where `y` is the integrand evaluated at each element of `x`. `fun` must be a function handle. Limits `a` and `b` can be `-Inf` or `Inf`. If both are finite, they can be complex. If at least one is complex, the integral is approximated over a straight line path from `a` to `b` in the complex plane.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`[q,errbnd] = quadgk(fun,a,b)` returns an approximate upper bound on the absolute error, `|Q - I|`, where `I` denotes the exact value of the integral.

`[q,errbnd] = quadgk(fun,a,b,param1,val1,param2,val2,...)` performs the integration with specified values of optional parameters. The available parameters are

Parameter	Description	
`'AbsTol'`	Absolute error tolerance. The default value of `'AbsTol'` is `1.e-10` (double), `1.e-5` (single).	`quadgk` attempts to satisfy `errbnd <= max(AbsTol,RelTol*|Q|)`. This is absolute error control when `|Q|` is sufficiently small and relative error control

Parameter	Description	
'RelTol'	Relative error tolerance. The default value of 'RelTol' is 1.e-6 (double), 1.e-4 (single).	when |Q| is larger. For pure absolute error control use 'AbsTol' > 0 and'RelTol'= 0. For pure relative error control use 'AbsTol' = 0. Except when using pure absolute error control, the minimum relative tolerance is 'RelTol' >= 100*eps(class(Q)).
'Waypoints'	Vector of integration waypoints.	If fun(x) has discontinuities in the interval of integration, the locations should be supplied as a Waypoints vector. When a, b, and the waypoints are all real, only the waypoints between a and b are used, and they are used in sorted order. Note that waypoints are not intended for singularities in fun(x). Singular points should be handled by making them endpoints of separate integrations and adding the results. If a, b, or any entry of the waypoints vector is complex, the integration is performed over a

Parameter	Description	
		sequence of straight line paths in the complex plane, from `a` to the first waypoint, from the first waypoint to the second, and so forth, and finally from the last waypoint to `b`.
`'MaxIntervalCount'`	Maximum number of intervals allowed. The default value is `650`.	The `'MaxIntervalCount'` parameter limits the number of intervals that `quadgk` uses at any one time after the first iteration. A warning is issued if `quadgk` returns early because of this limit. Routinely increasing this value is not recommended, but it may be appropriate when `errbnd` is small enough that the desired accuracy has nearly been achieved.

The list below contains information to help you determine which quadrature function in MATLAB to use:

- The `quad` function may be most efficient for low accuracies with nonsmooth integrands.
- The `quadl` function may be more efficient than `quad` at higher accuracies with smooth integrands.

- The quadgk function may be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The quadv function vectorizes quad for an array-valued fun.
- If the interval is infinite, [a,Inf), then for the integral of fun(x) to exist, fun(x) must decay as x approaches infinity, and quadgk requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but quadgk can be used if fun(x) decays fast enough.
- The quadgk function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint c like log|x-c| or $|x-c|^p$ for p >= -1/2. If the function is singular at points inside (a,b), write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with quadgk, and add the results.

Examples

Integrand with a singularity at an integration end point

Write a function myfun that computes the integrand:

```
function y = myfun(x)
y = exp(x).*log(x);
```

Then pass @myfun, a function handle to myfun, to quadgk, along with the limits of integration, 0 to 1:

```
q = quadgk(@myfun,0,1)

q =

   -1.3179
```

Alternatively, you can pass the integrand to quadgk as an anonymous function handle F:

```
f = (@(x)exp(x).*log(x));
q = quadgk(f,0,1);
```

Oscillatory integrand on a semi-infinite interval

Integrate over a semi-infinite interval with specified tolerances, and return the approximate error bound:

```
f = @(x)x.^5.*exp(-x).*sin(x);
[q,errbnd] = quadgk(f,0,inf,'RelTol',1e-8,'AbsTol',1e-12)

q =

  -15.0000

errbnd =

  9.4386e-009
```

Contour integration around a pole

Use `Waypoints` to integrate around a pole using a piecewise linear contour:

```
f = @(z)1./(2*z - 1);
q = quadgk(f,-1-i,-1-i,'Waypoints',[1-i,1+i,-1+i])

q =

   0.0000 + 3.1416i
```

Algorithms

`quadgk` implements adaptive quadrature based on a Gauss-Kronrod pair (15th and 7th order formulas).

Diagnostics

`quadgk` may issue one of the following warnings:

`'Minimum step size reached'` indicates that interval subdivision has produced a subinterval whose length is on the order of roundoff

error in the length of the original interval. A nonintegrable singularity is possible.

`'Reached the limit on the maximum number of intervals in use'` indicates that the integration was terminated before meeting the tolerance requirements and that continuing the integration would require more than `MaxIntervalCount` subintervals. The integral may not exist, or it may be difficult to approximate numerically. Increasing `MaxIntervalCount` usually does not help unless the tolerance requirements were nearly met when the integration was previously terminated.

`'Infinite or Not-a-Number function value encountered'` indicates a floating point overflow or division by zero during the evaluation of the integrand in the interior of the interval.

References

[1] L.F. Shampine "Vectorized Adaptive Quadrature in MATLAB," *Journal of Computational and Applied Mathematics*, 211, 2008, pp.131–140.

See Also

`quad2d` | `dblquad` | `quad` | `quadl` | `quadv` | `triplequad` | `function_handle` | `integral` | `integral2` | `integral3`

How To

- "Anonymous Functions"

quadl

Purpose

Numerically evaluate integral, adaptive Lobatto quadrature

Compatibility

`quadl` will be removed in a future release. Use `integral` instead.

Syntax

```
q = quadl(fun,a,b)
q = quadl(fun,a,b,tol)
quadl(fun,a,b,tol,trace)
[q,fcnt] = quadl(...)
```

Description

`q = quadl(fun,a,b)` approximates the integral of function `fun` from `a` to `b`, to within an error of 10^{-6} using recursive adaptive Lobatto quadrature. `fun` is a function handle. It accepts a vector `x` and returns a vector `y`, the function `fun` evaluated at each element of `x`. Limits `a` and `b` must be finite.

“Parameterizing Functions” explains how to provide additional parameters to the function `fun`, if necessary.

`q = quadl(fun,a,b,tol)` uses an absolute error tolerance of `tol` instead of the default, which is `1.0e-6`. Larger values of `tol` result in fewer function evaluations and faster computation, but less accurate results.

`quadl(fun,a,b,tol,trace)` with non-zero `trace` shows the values of `[fcnt a b-a q]` during the recursion.

`[q,fcnt] = quadl(...)` returns the number of function evaluations.

Use array operators `.*`, `./` and `.^` in the definition of `fun` so that it can be evaluated with a vector argument.

The function `quad` might be more efficient with low accuracies or nonsmooth integrands.

The list below contains information to help you determine which quadrature function in MATLAB to use:

- The `quad` function might be most efficient for low accuracies with nonsmooth integrands.

- The `quadl` function might be more efficient than `quad` at higher accuracies with smooth integrands.
- The `quadgk` function might be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The `quadv` function vectorizes `quad` for an array-valued `fun`.
- If the interval is infinite, `[a,Inf)`, then for the integral of `fun(x)` to exist, `fun(x)` must decay as `x` approaches infinity, and `quadgk` requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but `quadgk` can be used if `fun(x)` decays fast enough.
- The `quadgk` function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint `c` like `log|x-c|` or $|x-c|^p$ for `p >= -1/2`. If the function is singular at points inside `(a,b)`, write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with `quadgk`, and add the results.

Examples

Pass the function handle, `@myfun`, to `quadl`:

```
Q = quadl(@myfun,0,2);
```

where the function `myfun.m` is:

```
function y = myfun(x)
y = 1./(x.^3-2*x-5);
```

Pass anonymous function handle `F` to `quadl`:

```
F = @(x) 1./(x.^3-2*x-5);
Q = quadl(F,0,2);
```

Algorithms

`quadl` implements a high order method using an adaptive Gauss/Lobatto quadrature rule.

Diagnostics

`quadl` might issue one of the following warnings:

`'Minimum step size reached'` indicates that the recursive interval subdivision has produced a subinterval whose length is on the order of roundoff error in the length of the original interval. A nonintegrable singularity is possible.

`'Maximum function count exceeded'` indicates that the integrand has been evaluated more than 10,000 times. A nonintegrable singularity is likely.

`'Infinite or Not-a-Number function value encountered'` indicates a floating point overflow or division by zero during the evaluation of the integrand in the interior of the interval.

References

[1] Gander, W. and W. Gautschi, “Adaptive Quadrature – Revisited,” BIT, Vol. 40, 2000, pp. 84-101. This document is also available at `http://www.inf.ethz.ch/personal/gander`.

See Also

`quad2d` | `dblquad` | `quad` | `quadgk` | `triplequad` | `function_handle` | `integral` | `integral2` | `integral3`

How To

- “Anonymous Functions”

Purpose

Vectorized quadrature

Compatibility

`quadv` will be removed in a future release. Use `integral` with the `'ArrayValued'` option instead.

Syntax

```
Q = quadv(fun,a,b)
Q = quadv(fun,a,b,tol)
Q = quadv(fun,a,b,tol,trace)
[Q,fcnt] = quadv(...)
```

Description

`Q = quadv(fun,a,b)` approximates the integral of the complex array-valued function `fun` from `a` to `b` to within an error of `1.e-6` using recursive adaptive Simpson quadrature. `fun` is a function handle. The function `Y = fun(x)` should accept a scalar argument `x` and return an array result `Y`, whose components are the integrands evaluated at `x`. Limits `a` and `b` must be finite.

"Parameterizing Functions" explains how to provide addition parameters to the function `fun`, if necessary.

`Q = quadv(fun,a,b,tol)` uses the absolute error tolerance `tol` for all the integrals instead of the default, which is `1.e-6`.

Note The same tolerance is used for all components, so the results obtained with `quadv` are usually not the same as those obtained with `quad` on the individual components.

`Q = quadv(fun,a,b,tol,trace)` with non-zero `trace` shows the values of `[fcnt a b-a Q(1)]` during the recursion.

`[Q,fcnt] = quadv(...)` returns the number of function evaluations.

The list below contains information to help you determine which quadrature function in `MATLAB` to use:

- The `quad` function might be most efficient for low accuracies with nonsmooth integrands.

- The `quadl` function might be more efficient than `quad` at higher accuracies with smooth integrands.
- The `quadgk` function might be most efficient for high accuracies and oscillatory integrands. It supports infinite intervals and can handle moderate singularities at the endpoints. It also supports contour integration along piecewise linear paths.
- The `quadv` function vectorizes `quad` for an array-valued `fun`.
- If the interval is infinite, `[a,Inf)`, then for the integral of `fun(x)` to exist, `fun(x)` must decay as `x` approaches infinity, and `quadgk` requires it to decay rapidly. Special methods should be used for oscillatory functions on infinite intervals, but `quadgk` can be used if `fun(x)` decays fast enough.
- The `quadgk` function will integrate functions that are singular at finite endpoints if the singularities are not too strong. For example, it will integrate functions that behave at an endpoint `c` like `log|x-c|` or `|x-c|`^{p} for `p >= -1/2`. If the function is singular at points inside `(a,b)`, write the integral as a sum of integrals over subintervals with the singular points as endpoints, compute them with `quadgk`, and add the results.

Examples

For the parameterized array-valued function `myarrayfun`, defined by

```
function Y = myarrayfun(x,n)
Y = 1./((1:n)+x);
```

the following command integrates `myarrayfun`, for the parameter value n = 10 between a = 0 and b = 1:

```
Qv = quadv(@(x)myarrayfun(x,10),0,1);
```

The resulting array `Qv` has 10 elements estimating `Q(k) = log((k+1)./(k))`, for `k = 1:10`.

The entries in `Qv` are slightly different than if you compute the integrals using `quad` in a loop:

```
for k = 1:10
 Qs(k) = quadv(@(x)myscalarfun(x,k),0,1);
end
```

where myscalarfun is:

```
function y = myscalarfun(x,k)
y = 1./(k+x);
```

See Also

quad | quad2d | quadgk | quadl | dblquad | triplequad | function_handle | integral | integral2 | integral3

questdlg

Purpose

Create and open question dialog box

Syntax

```
button = questdlg('qstring')
button = questdlg('qstring','title')
button = questdlg('qstring','title',default)
button = questdlg('qstring','title','str1','str2',default)
button =  questdlg('qstring','title','str1','str2','str3',
   default)
button = questdlg('qstring','title', ..., options)
```

Description

`button = questdlg('qstring')` displays a modal dialog box presenting the question `'qstring'`. The dialog has three default buttons, **Yes**, **No**, and **Cancel**. If the user presses one of these three buttons, `button` is set to the name of the button pressed. If the user presses the close button on the dialog without making a choice, `button` is set to the empty string. If the user presses the **Return** key, button is set to `'Yes'`. `'qstring'` is a cell array or a string that automatically wraps to fit within the dialog box.

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

`button = questdlg('qstring','title')` displays a question dialog with `'title'` displayed in the dialog's title bar.

`button = questdlg('qstring','title',`*`default`*`)` specifies which push button is the default in the event that the **Return** key is pressed. `'default'` must be `'Yes'`, `'No'`, or `'Cancel'`.

`button = questdlg('qstring','title','str1','str2',`*`default`*`)` creates a question dialog box with two push buttons labeled `'str1'` and `'str2'`. *`default`* specifies the default button selection and must be `'str1'` or `'str2'`.

`button = questdlg('qstring','title','str1','str2','str3',default)` creates a question dialog box with three push buttons labeled `'str1'`, `'str2'`, and `'str3'`. *`default`* specifies the default button selection and must be `'str1'`, `'str2'`, or `'str3'`.

When *`default`* is specified, but is not set to one of the button names, pressing the **Enter** key displays a warning and the dialog remains open.

`button = questdlg('qstring','title', ..., options)` replaces the string *`default`* with a structure, `options`. The structure specifies which button string is the default answer, and whether to use TeX to interpret the question string, `qstring`. Button strings and dialog titles cannot use TeX interpretation. The `options` structure must include the fields `Default` and `Interpreter`, both strings. It can include other fields, but `questdlg` does not use them. You can set `Interpreter` to `'none'` or `'tex'`. If the `Default` field does not contain a valid button name, a command window warning is issued and the dialog box does not respond to pressing the **Enter** key.

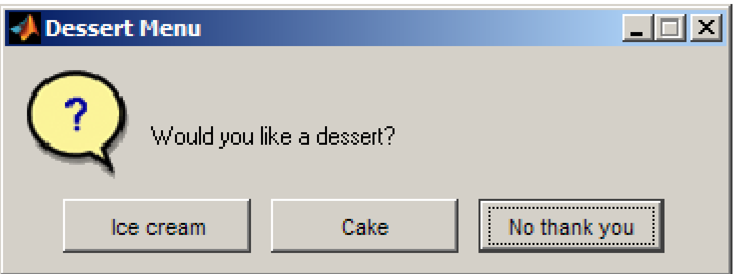
Examples

Example 1

Create a dialog that requests a dessert preference and encode the resulting choice as an integer.

```
% Construct a questdlg with three options
choice = questdlg('Would you like a dessert?', ...
 'Dessert Menu', ...
 'Ice cream','Cake','No thank you','No thank you');
% Handle response
switch choice
    case 'Ice cream'
        disp([choice ' coming right up.'])
        dessert = 1;
    case 'Cake'
        disp([choice ' coming right up.'])
        dessert = 2;
    case 'No thank you'
        disp('I''ll bring you your check.')
```

```
        dessert = 0;
end
```



To access the return value assigned to `dessert`, save the example as a function, for example `choosedessert`, by inserting this line on top:

```
function dessert = choosedessert
```

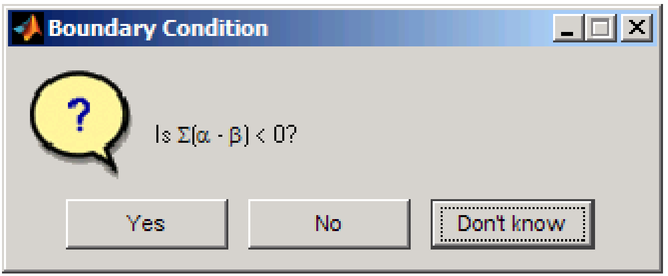
You can generalize the function by providing the cases as string or cell array calling arguments.

As the example shows, `case` statements can contain white space (but are case-sensitive).

Example 2

Specify an options structure to use the TeX interpreter to format a question.

```
options.Interpreter = 'tex';
% Include the desired Default answer
options.Default = 'Don''t know';
% Create a TeX string for the question
qstring = 'Is \Sigma(\alpha - \beta) < 0?';
choice = questdlg(qstring,'Boundary Condition',...
    'Yes','No','Don''t know',options)
```



See Also

dialog | errordlg | helpdlg | inputdlg | listdlg | msgbox | warndlg | figure | textwrap | uiwait | uiresume

quit

Purpose

Terminate MATLAB program

Alternatives

As an alternative to the quit function, use the Close box in the MATLAB desktop.

Syntax

```
quit
quit cancel
quit force
```

Description

`quit` displays a confirmation dialog box if the confirm upon quitting preference is selected, and if confirmed or if the confirmation preference is not selected, terminates MATLAB after running `finish.m`, if `finish.m` exists. The workspace is not automatically saved by `quit`. To save the workspace or perform other actions when quitting, create a `finish.m` file to perform those actions. For example, you can display a custom dialog box to confirm quitting using a `finish.m` file—see the following examples for details. If an error occurs while `finish.m` is running, `quit` is canceled so that you can correct your `finish.m` file without losing your workspace.

`quit` **`cancel`** is for use in `finish.m` and cancels quitting. It has no effect anywhere else.

`quit` **`force`** bypasses `finish.m` and terminates MATLAB. Use this to override `finish.m`, for example, if an errant `finish.m` will not let you quit.

Tips

When using Handle Graphics objects in `finish.m`, use `uiwait`, `waitfor`, or `drawnow` so that figures are visible. See the reference pages for these functions for more information.

If you want MATLAB to display the following confirmation dialog box after running quit, select **Preferences** in the **Environment** section on the **Home** tab. Then select the check box for Confirm before exiting MATLAB, and click **OK**.

Examples

Two sample `finish.m` files are included with MATLAB. Use them to help you create your own `finish.m`, or rename one of the files to `finish.m` to use it.

- `finishsav.m`—Saves the workspace to a MAT-file when MATLAB quits.
- `finishdlg.m`—Displays a dialog allowing you to cancel quitting; it uses `quit cancel` and contains the following code:

```
button = questdlg('Ready to quit?', ...
        'Exit Dialog','Yes','No','No');
switch button
            case 'Yes',
              disp('Exiting MATLAB');
              %Save variables to matlab.mat
              save
            case 'No',
              quit cancel;
end
```

See Also

`exit` | `save` | `finish` | `startup`

Quit (COM)

Purpose

Terminate MATLAB Automation server

Syntax

MATLAB Client

```
h.Quit
Quit(h)
invoke(h, 'Quit')
```

IDL Method Signature

```
void Quit(void)
```

Microsoft Visual Basic Client

```
Quit
```

Description

`Quit` terminates the MATLAB server session attached to handle `h`.

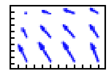
Tips

Server function names, like `Quit`, are case sensitive when using the first syntax shown.

There is no difference in the operation of the three syntaxes shown above for the MATLAB client.

Purpose

Quiver or velocity plot



Syntax

```
quiver(x,y,u,v)
quiver(u,v)
quiver(...,scale)
quiver(...,LineSpec)
quiver(...,LineSpec,'filled')
quiver(...,'PropertyName',PropertyValue,...)
quiver(axes_handle,...)
h = quiver(...)
```

Description

A quiver plot displays velocity vectors as arrows with components (u,v) at the points (x,y).

For example, the first vector is defined by components u(1),v(1) and is displayed at the point x(1),y(1).

quiver(x,y,u,v) plots vectors as arrows at the coordinates specified in each corresponding pair of elements in x and y. The matrices x, y, u, and v must all be the same size and contain corresponding position and velocity components. However, x and y can also be vectors, as explained in the next section. By default, the arrows are scaled to just not overlap, but you can scale them to be longer or shorter if you want.

quiver(u,v) draws vectors specified by u and v at equally spaced points in the *x-y* plane.

quiver(...,scale) automatically scales the arrows to fit within the grid and then stretches them by the factor scale. scale = 2 doubles their relative length, and scale = 0.5 halves the length. Use scale = 0 to plot the velocity vectors without automatic scaling. You can also tune the length of arrows after they have been drawn by choosing the **Plot Edit** tool, selecting the quivergroup object, opening the Property Editor, and adjusting the **Length** slider.

`quiver(...,LineSpec)` specifies line style, marker symbol, and color using any valid `LineSpec`. `quiver` draws the markers at the origin of the vectors.

`quiver(...,LineSpec,'filled')` fills markers specified by `LineSpec`.

`quiver(...,'PropertyName',PropertyValue,...)` specifies property name and property value pairs for the `quivergroup` objects the function creates.

`quiver(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = quiver(...)` returns the handle to the quivergroup object.

Expanding x- and y-Coordinates

MATLAB expands `x` and `y` if they are not matrices. This expansion is equivalent to calling `meshgrid` to generate matrices from vectors:

```
[x,y] = meshgrid(x,y);
quiver(x,y,u,v)
```

In this case, the following must be true:

`length(x) = n` and `length(y) = m`, where `[m,n] = size(u) = size(v)`.

The vector `x` corresponds to the columns of `u` and `v`, and vector `y` corresponds to the rows of `u` and `v`.

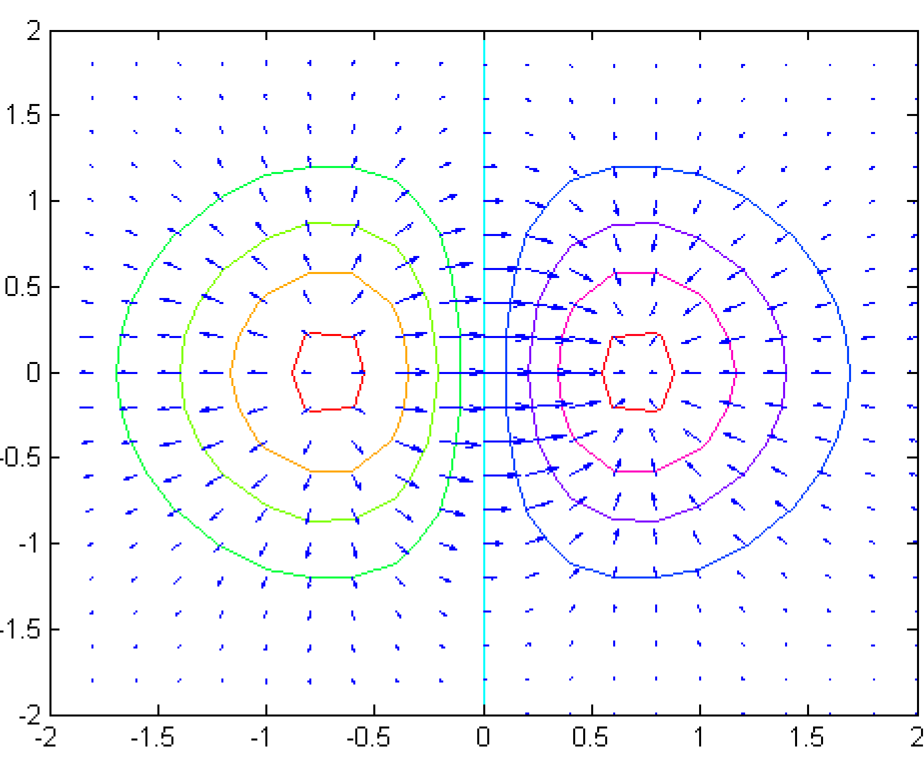
Examples

Showing the Gradient with Quiver Plots

Plot the gradient field of the function $z = xe^{(-x^2 - y^2)}$:

```
figure
[X,Y] = meshgrid(-2:.2:2);
Z = X.*exp(-X.^2 - Y.^2);
[DX,DY] = gradient(Z,.2,.2);
contour(X,Y,Z)
hold on
quiver(X,Y,DX,DY)
colormap hsv
```

```
hold off
```



See Also

`contour` | `LineSpec` | `plot` | `quiver3`

How To

- Two-Dimensional Quiver Plots

quiver3

Purpose

3-D quiver or velocity plot

Syntax

```
quiver3(x,y,z,u,v,w)
quiver3(z,u,v,w)
quiver3(...,scale)
quiver3(...,LineSpec)
quiver3(...,LineSpec,'filled')
quiver3(...,'PropertyName',PropertyValue,...)
quiver3(axes_handle,...)
h = quiver3(...)
```

Description

A three-dimensional quiver plot displays vectors with components (`u,v,w`) at the points (`x,y,z`), where `u,v,w,x,y,` and `z` all have real (non-complex) values.

`quiver3(x,y,z,u,v,w)` plots vectors with components (`u,v,w`) at the points (`x,y,z`). The matrices `x,y,z,u,v,w` must all be the same size and contain the corresponding position and vector components.

`quiver3(z,u,v,w)` plots the vectors at the equally spaced surface points specified by matrix `z`. `quiver3` automatically scales the vectors based on the distance between them to prevent them from overlapping.

`quiver3(...,scale)` automatically scales the vectors to prevent them from overlapping, and then multiplies them by `scale`. `scale = 2` doubles their relative length, and `scale = 0.5` halves them. Use `scale = 0` to plot the vectors without the automatic scaling.

`quiver3(...,LineSpec)` specifies line type and color using any valid `LineSpec`.

`quiver3(...,LineSpec,'filled')` fills markers specified by `LineSpec`.

`quiver3(...,'PropertyName',PropertyValue,...)` specifies property name and property value pairs for the `quivergroup` objects the function creates.

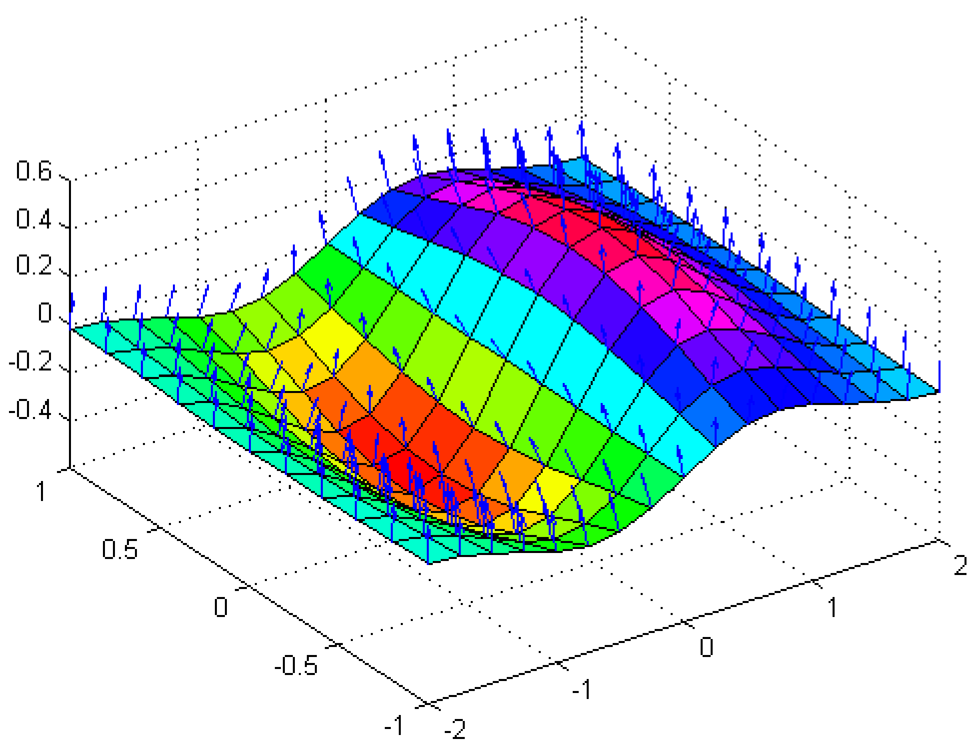
`quiver3(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes (`gca`).

`h = quiver3(...)` returns a vector of line handles.

Examples

Plot the surface normals of the function $z = xe^{(-x^2 - y^2)}$.

```
figure
[X,Y] = meshgrid(-2:0.25:2,-1:0.2:1);
Z = X.* exp(-X.^2 - Y.^2);
[U,V,W] = surfnorm(X,Y,Z);
quiver3(X,Y,Z,U,V,W,0.5);
hold on
surf(X,Y,Z);
colormap hsv
view(-35,45)
axis ([-2 2 -1 1 -.6 .6])
hold off
```



See Also

`axis` | `contour` | `LineSpec` | `plot` | `plot3` | `quiver` | `surfnorm` | `view`

How To

- Three-Dimensional Quiver Plots

Purpose

Define quivergroup properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands or the Property Editor (`propertyeditor`).

Note that you cannot define default properties for quivergroup objects.

See Plot Objects for more information on quivergroup objects.

Quivergroup Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`Annotation`

`hg.Annotation` object (read-only)

Control the display of quivergroup objects in legends. Specifies whether this quivergroup object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the quivergroup object is displayed in a figure legend.

IconDisplayStyle Value	Purpose
`on`	Include the quivergroup object in a legend as one entry, but not its children objects
`off`	Do not include the quivergroup or its children in a legend (default)
`children`	Include only the children of the quivergroup as separate entries in the legend

Setting the IconDisplayStyle Property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the IconDisplayStyle Property

See “Controlling Legends” for more information and examples.

`AutoScale`
`{on} | off`

Autoscale arrow length. Based on average spacing in the *x* and *y* directions, `AutoScale` scales the arrow length to fit within the grid-defined coordinate data and keeps the arrows from overlapping. After autoscaling, `quiver` applies the `AutoScaleFactor` to the arrow length.

`AutoScaleFactor`
scalar (default = `0.9`)

User-specified scale factor. When `AutoScale` is on, the `quiver` function applies this user-specified autoscale factor to the arrow length. A value of `2` doubles the length of the arrows; `0.5` halves the length.

`BeingDeleted`
`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object’s delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`
string | function handle

Button press callback function. Executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be:

- A string that is a valid MATLAB expression
- The name of a MATLAB file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`Children`
array of graphics object handles

Children of the quivergroup object. An array containing the handles of all line objects parented to this object (whether visible or not).

If a child object's `HandleVisibility` property is `callback` or `off`, its handle does not show up in this object's `Children` property. If you want the handle in the `Children` property, set the root `ShowHiddenHandles` property to `on`. For example:

```
set(0,'ShowHiddenHandles','on')
```

`Clipping`
`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs appear outside the axes plot box. This occurs if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`Color`
ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object's color. The default value is `[0 0 0]` (`black`).

See the `ColorSpec` reference page for more information on specifying color. See "Adding Arrows and Lines to Graphs".

`CreateFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback function executed during object creation. Defines a callback that executes when MATLAB creates an object. The default is an empty array.

You must specify the callback during the creation of the object. For example:

```
graphicfcn(y,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function and `graphicfcn` is the plotting function which creates this object.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DeleteFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback executed during object deletion. Executes when this object is deleted (for example, this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values. The default is an empty array.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`
string

String used by legend. The `legend` function uses the `DisplayName` property to label the quivergroup object in the legend. The default is an empty string.

- If you specify string arguments with the `legend` function, MATLAB set `DisplayName` to the corresponding string and uses that string for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, MATLAB set `DisplayName` to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.

- To add a legend programmatically that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See “Controlling Legends” for more information and examples.

`EraseMode`

`{normal} | none | xor | background`

Erase mode. Controls the technique MATLAB uses to draw and erase objects. Use alternative erase modes for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the object when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it because MATLAB stores no information about its former location.
- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object’s color depends on the color of whatever is beneath it on the display.
- `background` — Erase the object by redrawing it in the axes background `Color`, or the figure background `Color` if the axes `Color` property is `none`. This damages objects that are behind the erased object, but properly colors the erased object.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors (for example, performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible.
- `callback` — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching

the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties. See also `findall`.

Handle Validity

Hidden handles are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`

`{on} | off`

Selectable by mouse click. Determines whether this object can become the current object (as returned by the `gco` command and

the figure `CurrentObject` property) as a result of a mouse click on the line objects that compose the quiver plot. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`HitTestArea`
`on | {off}`

Select the object by clicking lines or area of extent. Select plot objects by:

- Clicking quiver arrows (default).
- Clicking anywhere in the extent of the plot.

When `HitTestArea` is `off`, you must click the quiver lines (excluding the baseline) to select the object. When `HitTestArea` is `on`, you can select this object by clicking anywhere within the extent of the plot (that is, anywhere within a rectangle that encloses all the arrows).

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the `Interruptible` property affects only the callbacks for the`ButtonDownFcn` property. A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both the callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see “Control Callback Execution and Interruption”.

`LineStyle`

`{-} | -- | : | -. | none`

Line style of quivergroup object.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line
'none'	No line

Use `LineStyle` `none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`
size in points

Width of linear objects and edges of filled areas. Specify in points. 1 point = $^1/_{72}$ inch. The default is `0.5` points.

`Marker`
character (see table)

Marker symbol. Specifies marks that display at data points. You can set values for the `Marker` property independently from the `LineStyle` property. For a list of supported marker symbols, see the following table.

Marker Specifiers Table

Specifier	Marker Type
'+'	Plus sign
'o'	Circle
'*'	Asterisk

Specifier	Marker Type
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'''	Six-pointed star (hexagram)
'none'	No marker (default)

`MarkerEdgeColor`

`ColorSpec | none | {auto}`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `ColorSpec` — User-defined color.
- `none` — Specifies no color, which makes nonfilled markers invisible.
- `auto` — Uses same color as the `Color` property.

`MarkerFaceColor`

`ColorSpec | {none} | auto`

Fill color for closed-shape markers. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `ColorSpec` — User-defined color.
- `none` — Makes the interior of the marker transparent, allowing the background to show through.
- `auto` — Sets the fill color to the axes `Color` property. If the axes `Color` property is `none`, sets the fill color to the figure `Color`.

`MarkerSize`
scalar

Marker size. Size of the marker in points. The default value is `6`.

Note MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`MaxHeadSize`
scalar (default = `0.2`

Maximum size of arrowhead. A value determining the maximum size of the arrowhead relative to the length of the arrow.

`Parent`
handle of parent axes, hggroup, or hgtransform

Parent of object. Handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Object selection state. When you set this property to `on`, MATLAB displays selection handles at the corners and midpoints if the

`SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
`{on} | off`

Object highlighted when selected.

- `on` — MATLAB indicates the selected state by drawing four edge handles and four corner handles.
- `off` — MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`ShowArrowHead`
`{on} | off`

Display arrowheads on vectors. When this property is `on`, MATLAB draws arrowheads on the vectors displayed by `quiver`. When you set this property to off, `quiver` draws the vectors as lines without arrowheads.

`Tag`
string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

For example, create an areaseries object and set the `Tag` property.

```
t = area(Y,'Tag','area1')
```

When you want to access objects of a given type, use `findobj` to find the object's handle. The following statement changes the `FaceColor` property of the object whose `Tag` is `area1`.

```
set(findobj('Tag','area1'),'FaceColor','red')
```

`Type`
string (read-only)

Type of graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For stem objects, `Type` is '`hggroup`'. This statement finds all the hggroup objects in the current axes.

```
t = findobj(gca,'Type','hggroup');
```

`UIContextMenu`
handle of uicontextmenu object

Associate context menu with object. Handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object. The default value is an empty array.

`UserData`
array

User-specified data. Data you want to associate with this object (including cell arrays and structures). The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`
`{on} | off`

Visibility of object and its children.

- `on` — Object and all children of the object are visible unless the child object's `Visible` property is `off`.
- `off` — Object not displayed. However, the object still exists and you can set and query its properties.

`UData`

matrix

One dimension of 2-D or 3-D vector components. `UData`, `VData` and `WData` (for 3-D) together specify the components of the vectors displayed as arrows in the quiver graph. For example, the first vector is defined by components `UData(1)`,`VData(1)`,`WData(1)`.

`UDataSource`

string (MATLAB variable)

Link `UData` *to MATLAB variable.* Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the `UData`. The default value is an empty array.

```
set(h,'UDataSource','UDatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `UDataSource` does not change the object's `UData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`VData`

matrix

One dimension of 2-D or 3-D vector components. `UData`, `VData` and `WData` (for 3-D) together specify the components of the vectors displayed as arrows in the quiver graph. For example, the first vector is defined by components `UData(1)`,`VData(1)`,`WData(1)`.

`VDataSource`

MATLAB variable, as a string

Link `VData` to MATLAB variable. Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the `VData`. The default value is an empty array.

```
set(h,'VDataSource','VDatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `VDataSource` does not change the object's `VData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`WData`

matrix

One dimension of 2-D or 3-D vector components. `UData`, `VData` and `WData` (for 3-D) together specify the components of the vectors

displayed as arrows in the quiver graph. For example, the first vector is defined by components `UData(1)`,`VData(1)`,`WData(1)`.

`WDataSource`

MATLAB variable, as a string

Link `WData` to MATLAB variable. Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the `WData`. The default value is an empty array.

```
set(h,'WDataSource','WDatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `WDataSource` does not change the object's `WData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`XData`

vector | matrix

X-axis coordinates of arrows. The `quiver` function draws an individual arrow at each *x*-axis location in the `XData` array.`XData` can be either a matrix equal in size to all other data properties or for 2-D, a vector equal in length to the number of columns in `UData` or `VData`. That is, `length(XData) == size(UData,2)`.

If you do not specify `XData` (which is the input argument `X`), the `quiver` function uses the indices of `UData` to create the quiver graph. See the `XDataMode` property for related information.

`XDataMode`
`{auto} | manual`

Use automatic or user-specified x-axis values. If you specify `XData` (by setting the `XData` property or specifying the input argument `X`), the `quiver` function sets this property to `manual`.

If you set `XDataMode` to `auto` after having specified `XData`, the `quiver` function resets the *x* tick-mark labels to the indices of the `U`, `V`, and `W` data, overwriting any previous values.

`XDataSource`
MATLAB variable, as a string

Link `XData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `XData`. The default value is an empty array.

```
set(h,'XDataSource','xdatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `XDataSource` does not change the object's `XData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`YData`

vector | matrix

Y-axis coordinates of arrows. The `quiver` function draws an individual arrow at each *y*-axis location in the `YData` array. `YData` can be either a matrix equal in size to all other data properties or for 2-D, a vector equal in length to the number of rows in `UData` or `VData`. That is, `length(YData) == size(UData,1)`.

If you do not specify `YData` (which is the input argument `Y`), the `quiver` function uses the indices of `VData` to create the quiver graph. See the `YDataMode` property for related information.

The input argument `Y` in the `quiver` function calling syntax assigns values to `YData`.

`YDataMode`

`{auto} | manual`

Use automatic or user-specified y-axis values. If you specify `YData` (by setting the `YData` property or specifying the input argument `Y`), MATLAB sets this property to `manual`.

If you set `YDataMode` to auto after having specified `YData`, MATLAB resets the *y* tick-mark labels to the indices of the `U`, `V`, and `W` data, overwriting any previous values.

`YDataSource`

MATLAB variable, as a string

Link `YData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `YData`. The default value is an empty array.

```
set(h,'YDataSource','Ydatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's

`YDataSource` does not change the object's `YData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`ZData`

vector | matrix

Z-axis coordinates of arrows. The `quiver` function draws an individual arrow at each *z*-axis location in the `ZData` array. `ZData` must be a matrix equal in size to `XData` and `YData`.

The input argument `Z` in the `quiver3` function calling syntax assigns values to `ZData`.

Purpose

QZ factorization for generalized eigenvalues

Syntax

```
[AA,BB,Q,Z] = qz(A,B)
[AA,BB,Q,Z,V,W] = qz(A,B)
qz(A,B,flag)
```

Description

The `qz` function gives access to intermediate results in the computation of generalized eigenvalues.

`[AA,BB,Q,Z] = qz(A,B)` for square matrices `A` and `B`, produces upper quasitriangular matrices `AA` and `BB`, and unitary matrices `Q` and `Z` such that `Q*A*Z = AA`, and `Q*B*Z = BB`. For complex matrices, `AA` and `BB` are triangular.

`[AA,BB,Q,Z,V,W] = qz(A,B)` also produces matrices `V` and `W` whose columns are generalized eigenvectors.

`qz(A,B,flag)` for real matrices `A` and `B`, produces one of two decompositions depending on the value of `flag`:

`'complex'`	Produces a possibly complex decomposition with a triangular `AA`. For compatibility with earlier versions, `'complex'` is the default.
`'real'`	Produces a real decomposition with a quasitriangular `AA`, containing 1-by-1 and 2-by-2 blocks on its diagonal.

If `AA` is triangular, the diagonal elements of `AA` and `BB`, $\alpha = \text{diag}(AA)$ and $\beta = \text{diag}(BB)$, are the generalized eigenvalues that satisfy

$$A * V * \beta = B * V * \alpha$$
$$\beta' * W' * A = \alpha' * W' * B$$

The eigenvalues produced by

$$\lambda = \text{eig}(A, B)$$

are the ratios of the αs and ßs.

$$\lambda = \alpha./\beta$$

If AA is not triangular, it is necessary to further reduce the 2-by-2 blocks to obtain the eigenvalues of the full system.

See Also

eig

Purpose

Uniformly distributed pseudorandom numbers

Syntax

```
r = rand(n)
r = rand(m,n)
r = rand([m,n])
r = rand(m,n,p,...)
r = rand([m,n,p,...])
r = rand
r = rand(size(A))
r = rand(..., 'double')
r = rand(..., 'single')
```

Description

`r = rand(n)` returns an `n`-by-`n` matrix containing pseudorandom values drawn from the standard uniform distribution on the open interval (0,1). `r = rand(m,n)` or `r = rand([m,n])` returns an `m`-by-`n` matrix. `r = rand(m,n,p,...)` or `r = rand([m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-... array. `r = rand` returns a scalar. `r = rand(size(A))` returns an array the same size as `A`.

`r = rand(..., 'double')` or `r = rand(..., 'single')` returns an array of uniform values of the specified class.

Note The size inputs `m`, `n`, `p`, ... should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by `rand` is determined by the internal settings of the uniform random number generator that underlies `rand`, `randi`, and `randn`. You can control that shared random number generator using `rng`.

Note To use the `rng` function instead of `rand` or `randn` with the `'seed'`, `'state'`, or `'twister'` inputs, see the documentation on “Updating Your Random Number Generator Syntax”

Examples

Example 1

Generate values from the uniform distribution on the interval [a, b]:

```
r = a + (b-a).*rand(100,1);
```

Example 2

Use the randi function, instead of rand, to generate integer values from the uniform distribution on the set 1:100:

```
r = randi(100,1,5);
```

Example 3

Reset the random number generator used by rand, randi, and randn to its default startup settings, so that rand produces the same random numbers as if you restarted MATLAB:

```
rng('default')
rand(1,5)
ans =
    0.8147    0.9058    0.1270    0.9134    0.6324
```

Example 4

Save the settings for the random number generator used by rand, randi, and randn, generate 5 values from rand, restore the settings, and repeat those values:

```
s = rng;
u1 = rand(1,5)
u1 =
    0.0975    0.2785    0.5469    0.9575    0.9649

rng(s);
u2 = rand(1,5)
u2 =
    0.0975    0.2785    0.5469    0.9575    0.9649
```

u2 contains exactly the same values as u1.

Example 5

Reinitialize the random number generator used by `rand`, `randi`, and `randn` with a seed based on the current time. `rand` returns different values each time you do this. Note that it is usually not necessary to do this more than once per MATLAB session as it may affect the statistical properties of the random numbers MATLAB produces:

```
rng('shuffle');
rand(1,5);
```

See Also

`randi` | `randn` | `rng` | `@RandStream` | `rand (RandStream)` | `sprand` | `sprandn` | `randperm`

rand (RandStream)

Purpose

Uniformly distributed random numbers

Class

`RandStream`

Syntax

```
r = rand(s,n)
rand(s,m,n)
rand(s,[m,n])
rand(s,m,n,p,...)
rand(s,[m,n,p,...])
rand(s)
rand(s,size(A))
r = rand(..., 'double')
r = rand(..., 'single')
```

Description

`r = rand(s,n)` returns an `n`-by-`n` matrix containing pseudorandom values drawn from the standard uniform distribution on the open interval (0,1). The values are drawn from the random stream `s`. `rand(s,m,n)` or `rand(s,[m,n])` returns an `m`-by-`n` matrix. `rand(s,m,n,p,...)` or `rand(s,[m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-... array. `rand(s)` returns a scalar. `rand(s,size(A))` returns an array the same size as `A`.

`r = rand(..., 'double')` or `r = rand(..., 'single')` returns an array of uniform values of the specified class.

Note The size inputs `m`, `n`, `p`, ... should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by `rand` is determined by the internal state of the random number stream `s`. Resetting that stream to the same fixed state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties.

See Also

`rand` | `@RandStream` | `randi (RandStream)` | `randn (RandStream)` | `randperm (RandStream)`

randi

Purpose

Uniformly distributed pseudorandom integers

Syntax

```
r = randi(imax,n)
r = randi(imax,m,n)
r = randi(imax,[m,n])
r = randi(imax,p1,...,pn)
r = randi(imax,[p1,...,pn])
r = randi(imax)
r = randi(imax,size(A))
r = randi([imin,imax],...)
r = randi(..., classname)
```

Description

`r = randi(imax,n)` returns an `n`-by-`n` matrix containing pseudorandom integer values drawn from the discrete uniform distribution on the interval [`1`,`imax`].

`r = randi(imax,m,n)` and `r = randi(imax,[m,n])` return an `m`-by-`n` matrix of numbers. Both `m` and `n` must be nonnegative integer values. If either `m` or `n` is negative, then it is treated as `0`.

`r = randi(imax,p1,...,pn)` and `r = randi(imax,[p1,...,pn])` return an `n`-dimensional array.

`r = randi(imax)` returns a scalar value between `1` and `imax`. This is the same as `r = randi(imax,1)`

`r = randi(imax,size(A))` returns an array the same size as `A`.

`r = randi([imin,imax],...)` returns an array containing integer values drawn from the discrete uniform distribution on the interval [`imin`,`imax`].

`r = randi(..., classname)` returns an array of integer values of class `classname`. `classname` does not support 64-bit integers.

Specify the size arguments, `m`, `n` or `p1`,..., `pn`, as nonnegative integers. If you pass a negative size value, it will be treated as 0.

If you specify a distribution range using two numbers, `[imin,imax]`, both numbers must be integers that satisfy `imin` ≤ `imax`. If you specify

the range using only the upper bound, `imax`, it must be a positive integer (greater than zero).

The sequence of numbers produced by `randi` is determined by the settings of the uniform random number generator that underlies `rand`, `randn`, and `randi`. `randi` uses one uniform random value to create each integer random value. You can control that shared random number generator using `rng`.

Examples

Example 1

Generate a `100`-by-`1` array of integer values from the uniform distribution on the set 1:10:

```
r = randi(10,100,1);
```

Example 2

Generate a `100`-by-`1` array of integers drawn uniformly from 1:10:

```
r = randi(10,100,1,'uint32');
```

Example 3

Generate a `100`-by-`1` array of integer values drawn uniformly from -10:10:

```
r = randi([-10 10],100,1);
```

Example 4

Reset the random number generator used by `rand`, `randi`, and `randn` to its default startup settings, so that `randi` produces the same random numbers as if you restarted MATLAB:

```
rng('default');
randi(10,1,5)

ans =
     9    10     2    10     7
```

Example 5

Save the settings for the random number generator used by rand, randi, and randn, generate 5 values from randi, restore the settings, and repeat those values:

```
s = rng;
i1 = randi(10,1,5)
i1 =
     1     3     6    10    10
rng(s);
i2 = randi(10,1,5)
i2 =
     1     3     6    10    10
```

i2 contains exactly the same values as i1.

Example 6

Reinitialize the random number generator used by rand, randi, and randn with a seed based on the current time. randi returns different values each time you do this. Note that it is usually not necessary to do this more than once per MATLAB session:

```
rng('shuffle');
randi(10,1,5);
```

See Also

rand | randn | rng | @RandStream | randi (RandStream)

Purpose

Uniformly distributed pseudorandom integers

Class

`RandStream`

Syntax

```
r = randi(s,imax,n)
randi(s,imax,m,n)
randi(s,imax,[m,n])
randi(s,imax,m,n,p,...)
randi(s,imax,[m,n,p,...])
randi(s,imax)
randi(s,imax,size(A))
r = randi(s,[imin,imax],...)
r = randi(..., classname)
```

Description

`r = randi(s,imax,n)` returns an `n`-by-`n` matrix containing pseudorandom integer values drawn from the discrete uniform distribution on `1:imax`. `randi` draws those values from the random stream `s`. `randi(s,imax,m,n)` or `randi(s,imax,[m,n])` returns an `m`-by-`n` matrix. `randi(s,imax,m,n,p,...)` or `randi(s,imax,[m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-... array. `randi(s,imax)` returns a scalar. `randi(s,imax,size(A))` returns an array the same size as `A`.

`r = randi(s,[imin,imax],...)` returns an array containing integer values drawn from the discrete uniform distribution on `imin:imax`.

`r = randi(..., classname)` returns an array of integer values of class `classname`. `classname` does not support 64-bit integers.

Note The size inputs `m`, `n`, `p`, ... should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by `randi` is determined by the internal state of the random stream `s`. `randi` uses one uniform value from `s` to generate each integer value. Resetting `s` to the same fixed

state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties.

See Also

`randi` | `RandStream` | `rand (RandStream)` | `randn (RandStream)`

Purpose

Normally distributed pseudorandom numbers

Syntax

```
r = randn(n)
r = randn(m,n)
r = randn([m,n])
r = randn(m,n,p,...)
r = randn([m,n,p,...])
r = randn
r = randn(size(A))
r = randn(..., 'double')
r = randn(..., 'single')
```

Description

`r = randn(n)` returns an `n`-by-`n` matrix containing pseudorandom values drawn from the standard normal distribution. `r = randn(m,n)` or `r = randn([m,n])` returns an `m`-by-`n` matrix. `r = randn(m,n,p,...)` or `r = randn([m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-... array. `r = randn` returns a scalar. `r = randn(size(A))` returns an array the same size as `A`.

`r = randn(..., 'double')` or `r = randn(..., 'single')` returns an array of normal values of the specified class.

Note The size inputs `m`, `n`, `p`, ... should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by `randn` is determined by the settings of the uniform random number generator that underlies `rand`, `randn`, and `randi`. `randn` uses one or more uniform random values to create each normal random value. You can control that shared random number generator using `rng`.

Note To use the `rng` function instead of `rand` or `randn` with the `'seed'`, `'state'`, or `'twister'` inputs, see the documentation on "Updating Your Random Number Generator Syntax"

Examples

Example 1

Generate values from a normal distribution with mean 1 and standard deviation 2:

```
r = 1 + 2.*randn(100,1);
```

Example 2

Generate values from a bivariate normal distribution with specified mean vector and covariance matrix:

```
mu = [1 2];
Sigma = [1 .5; .5 2]; R = chol(Sigma);
z = repmat(mu,100,1) + randn(100,2)*R;
```

Example 3

Reset the random number generator used by `rand`, `randi`, and `randn` to its default startup settings, so that `randn` produces the same random numbers as if you restarted MATLAB:

```
rng('default');
randn(1,5)
ans =
    0.5377    1.8339   -2.2588    0.8622    0.3188
```

Example 4

Save the settings for the random number generator used by `rand`, `randi`, and `randn`, generate 5 values from `randn`, restore the settings, and repeat those values:

```
s = rng;
```

```
z1 = randn(1,5)
z1 =
   -1.3077   -0.4336    0.3426    3.5784    2.7694
rng(s);
z2 = randn(1,5)
z2 =
   -1.3077   -0.4336    0.3426    3.5784    2.7694
```

`z2` contains exactly the same values as `z1`.

Example 5

Reinitialize the random number generator used by `rand`, `randi`, and `randn` with a seed based on the current time. `randn` returns different values each time you do this. Note that it is usually not necessary to do this more than once per MATLAB session:

```
rng('shuffle');
randn(1,5)
```

See Also

`rand` | `randi` | `rng` | `@RandStream` | `randn (RandStream)`

randn (RandStream)

Purpose Normally distributed pseudorandom numbers

Class `RandStream`

Syntax

```
randn(s,m,n)
randn(s,[m,n])
randn(s,m,n,p,...)
randn(s,[m,n,p,...])
randn(s)
randn(s,size(A))
r = randn(..., 'double')
r = randn(..., 'single')
```

Description

`r = randn(s,n)` returns an `n`-by-`n` matrix containing pseudorandom values drawn from the standard normal distribution. `randn` draws those values from the random stream s. `randn(s,m,n)` or `randn(s,[m,n])` returns an `m`-by-`n` matrix. `randn(s,m,n,p,...)` or `randn(s,[m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-... array. `randn(s)` returns a scalar. `randn(s,size(A))` returns an array the same size as `A`.

`r = randn(..., 'double')` or `r = randn(..., 'single')` returns an array of uniform values of the specified class.

Note The size inputs `m`, `n`, `p`, ... should be nonnegative integers. Negative integers are treated as 0.

The sequence of numbers produced by `randn` is determined by the internal state of the random stream `s`. `randn` uses one or more uniform values from `s` to generate each normal value. Resetting that stream to the same fixed state allows computations to be repeated. Setting the stream to different states leads to unique computations, however, it does not improve any statistical properties.

See Also `randn` | `RandStream` | `rand (RandStream)` | `randi (RandStream)`

Purpose

Random permutation

Syntax

```
p = randperm(n)
p = randperm(n,k)
```

Description

`p = randperm(n)` returns a row vector containing a random permutation of the integers from 1 to `n` inclusive.

`p = randperm(n,k)` returns a row vector containing `k` unique integers selected randomly from 1 to `n` inclusive.

Tips

For `p = randperm(n,k)`, `p` contains `k` *unique* values. `randperm` performs k-permutations (sampling without replacement). To allow repeated values in the output (sampling with replacement), use `randi(n,1,k)`.

`randperm` uses the same random number generator as `rand`, `randi`, and `randn`. You control this generator with `rng`.

Examples

```
randperm(6)
```

might be the vector

```
[3  2  6  4  1  5]
```

or it might be some other permutation of the integers from 1 to 6, depending on the state of the random number generator. Two successive calls to `randperm` would in most cases return two different vectors:

```
randperm(6)
ans =
     5     2     6     4     1     3

randperm(6)
ans =
     4     1     6     2     3     5
```

```
randperm(6,3)
```

might be the vector

```
[4 2 5]
```

or it might be some other permutation of any three integers from 1 to 6 inclusive, depending on the state of the random number generator.

See Also

`permute` | `nchoosek` | `randi` | `randperm(RandStream)` | `perms` | `rng`

Purpose

Random permutation

Class

`RandStream`

Syntax

```
p = randperm(s,n)
p = randperm(s,n,k)
```

Description

`p = randperm(s,n)` returns a row vector containing a random permutation of integers from 1 to `n` inclusive. `randperm(s,n)` uses random values drawn from the random stream `s`.

`p = randperm(s,n,k)` returns a row vector containing `k` unique integers selected randomly from 1 to `n` inclusive.

Tips

For `p = randperm(s,n,k)`, `p` contains `k` *unique* values. `randperm` performs k-permutations (sampling without replacement). To allow repeated values in the output (sampling with replacement), use `randi(s,n,1,k)`.

Examples

Create a random stream `s` and generate a random permutation of the integers from 1 to 6 based on `s`:

```
s = RandStream('mt19937ar','Seed',0);
randperm(s,6)
```

MATLAB returns the vector

```
[6     3     5     1     2     4]
```

Use the random stream `s` to generate three integers between 1 and 10:

```
randperm(s,10,3)
ans =
      1     8     9
```

See Also

`permute` | `randperm` | `nchoosek` | `perms` | `rand` | `randi (RandStream)`

Purpose Random number stream

Constructor `RandStream`

Description Pseudorandom numbers in MATLAB come from one or more random number streams. The simplest way to generate arrays of random numbers is to use `rand`, `randn`, or `randi`. These functions all rely on the same stream of uniform random numbers, known as the *global stream*. You can create other streams that act separately from the global stream, and you can use their `rand`, `randi`, or `randn` methods to generate arrays of random numbers. You can also create a random number stream and make it the global stream.

To create a single random number stream, use the `RandStream` constructor. To create multiple independent random number streams, use `RandStream.create`. The `rng` function provides a simple interface to create a new global stream.

`stream = RandStream.getGlobalStream` returns the global random number stream, that is, the one currently used by the `rand`, `randi`, and `randn` functions.

`prevstream = RandStream.setGlobalStream(stream)` designates the random number stream `stream` as the new global stream to be used by the `rand`, `randi`, and `randn` functions, and returns the previous global stream.

A random number stream `s` has properties that control its behavior. Access or assign to a property using `p = s.Property` or `s.Property = p`. The following table lists defined properties:

Properties

Property	Description
`Type`	(Read-only) Generator algorithm used by the stream. The list of possible generators is given by `RandStream.list`.
`Seed`	(Read-only) Seed value used to create the stream.
`NumStreams`	(Read-only) Number of streams in the group in which the current stream was created.
`StreamIndex`	(Read-only) Index of the current stream from among the group of streams with which it was created.
`State`	Internal state of the generator. You should not depend on the format of this property. The value you assign to `S.State` must be a value read from `S.State` previously. Use `reset` to return a stream to a predictable state without having previously read from the `State` property. The sequence of random numbers produced by a random number stream `s` is determined by the internal state of its random number generator. Saving and restoring the generator's internal state with the `State` property allows you to reproduce a sequence of random numbers.

Property	Description
Substream	Index of the substream to which the stream is currently set. The default is 1. Multiple substreams are not supported by all generator types; the multiplicative lagged Fibonacci generator (mlfg6331_64) and combined multiple recursive generator (mrg32k3a) support substreams.
NormalTransform	Transformation algorithm used by randn(s, ...) to generate normal pseudorandom values. Possible values are 'Ziggurat', 'Polar', or 'Inversion'.
RandnAlg	RandnAlg has been removed. Use NormalTransform instead. Transformation algorithm that will be used by randn(S, ...) to generate normal pseudorandom values. Options are 'Ziggurat', 'Polar', or 'Inversion'.
Antithetic	Logical value indicating whether S generates antithetic pseudorandom values. For uniform values, these are the usual values subtracted from 1. The default is false.
FullPrecision	Logical value indicating whether S generates values using its full precision. Some generators can create pseudorandom values faster, but with fewer random bits, if FullPrecision is false. The default is true.

Methods

Method	Description
`RandStream`	Create a random number stream.
`RandStream.create`	Create multiple independent random number streams.
`get`	Get the properties of a random stream object.
`list`	List available random number generator algorithms.
`set`	Set random stream property.
`RandStream.getGlobalStream`	Get the global random number stream.
`RandStream.getDefaultStream`	`RandStream.getDefaultStream` has been removed. Use `RandStream.getGlobalStream` instead. The shared random number stream used by `rand`, `randi`, and `randn` is now referred to as the global stream. Default random number stream.
`RandStream.setGlobalStream`	Set global random number stream.
`RandStream.setDefaultStream`	`RandStream.setDefaultStream` has been removed. Use `RandStream.setGlobalStream` instead. Set default random number stream property.
`reset`	Reset a stream to its initial internal state
`rand`	Pseudorandom numbers from a uniform distribution

Method	Description
randn	Pseudorandom numbers from a standard normal distribution
randi	Pseudorandom integers from a uniform discrete distribution
randperm	Random permutation of a set of values

Examples

Example 1

Create a single stream and designate it as the current global stream:

```
s = RandStream('mt19937ar','Seed',1);
RandStream.setGlobalStream(s);
```

Example 2

Create three independent streams:

```
[s1,s2,s3] = RandStream.create('mrg32k3a','NumStreams',3);
r1 = rand(s1,100000,1);
r2 = rand(s2,100000,1);
r3 = rand(s3,100000,1);
corrcoef([r1,r2,r3])
```

Example 3

Create only one stream from a set of three independent streams, and designate it as the current global stream:

```
s2 = RandStream.create('mrg32k3a','NumStreams',3,...
     'StreamIndices',2);
RandStream.setGlobalStream(s2);
```

Example 4

Reset the global random number stream that underlies `rand`, `randi`, and `randn` back to its beginning, to reproduce previous results:

```
stream = RandStream.getGlobalStream;
```

```
reset(stream);
```

Example 5

Save and restore the current global stream's state to reproduce the output of `rand`:

```
stream = RandStream.getGlobalStream;
savedState = stream.State;
u1 = rand(1,5)
u1 =
    0.8147    0.9058    0.1270    0.9134    0.6324

stream.State = savedState;
u2 = rand(1,5)
u2 =
    0.8147    0.9058    0.1270    0.9134    0.6324
```

`u2` contains exactly the same values as `u1`.

Example 6

Reset the global random number stream to its initial settings. This causes `rand`, `randi`, and `randn` to start over, as if in a new MATLAB session:

```
s = RandStream('mt19937ar','Seed',0);
RandStream.setGlobalStream(s);
```

Example 7

Reinitialize the global random number stream using a seed based on the current time. This causes `rand`, `randi`, and `randn` to return different values in different MATLAB sessions. It is usually not desirable to do this more than once per MATLAB session as it may affect the statistical properties of the random numbers MATLAB produces:

```
s = RandStream('mt19937ar','Seed','shuffle');
RandStream.setGlobalStream(s);
```

Example 8

Change the transformation algorithm that `randn` uses to create normal pseudorandom values from uniform values. This does not replace or reset the global stream.

```
stream = RandStream.getGlobalStream;
stream.NormalTransform = 'inversion'
```

See Also

`rand` | `rng` | `randn` | `randi`

Purpose

Random number stream

Class

`RandStream`

Syntax

```
s = RandStream('gentype')
s = RandStream('gentype',Name,Value)
```

Description

`s = RandStream('gentype')` creates a random number stream that uses the uniform pseudorandom number generator algorithm specified by `gentype`. `RandStream.list` returns all possible values for `gentype`, or see “Choosing a Random Number Generator” for details on generator algorithms.

`s = RandStream('gentype',Name,Value)` allows you to specify one or more optional `Name,Value` pairs to control creation of the stream.

Once you have created a random, you can use `RandStream.setGlobalStream` to make it the global stream, so that the functions `rand`, `randi`, and `randn` draw values from it.

Parameters are for `RandStream` are:

Parameter	Description
`Seed`	Nonnegative scalar integer with which to initialize all streams. Seeds must be an integer between 0 and $2^{32} - 1$ or `'shuffle'` to create a seed based on the current time. Default is 0.
`NormalTransform`	Transformation algorithm used by `randn(s, ...)` to generate normal pseudorandom values. Possible values are `'Ziggurat'`, `'Polar'`, or `'Inversion'`.
`RandnAlg`	`RandnAlg` has been removed. Use `NormalTransform` instead.

Parameter	Description
	Transformation algorithm used by `randn` to generate normal pseudorandom values. Options are `'Ziggurat'`, `'Polar'`, or `'Inversion'`.

Examples

Example 1

Create a random number stream, make it the global stream, and save and restore its state to reproduce the output of `randn`:

```
s = RandStream('mrg32k3a');
RandStream.setGlobalStream(s);
savedState = s.State;
z1 = randn(1,5)
z1 =
   -0.1894   -1.4426   -0.3592    0.8883   -0.4337
s.State = savedState;
z2 = randn(1,5)
z2 =
   -0.1894   -1.4426   -0.3592    0.8883   -0.4337
```

`z2` contains exactly the same values as `z1`.

Example 2

Return `rand`, `randi`, and `randn` to their default startup settings:

```
s = RandStream('mt19937ar','Seed',0)
RandStream.setGlobalStream(s);
```

Example 3

Replace the current global random number stream with a stream whose seed is based on the current time, so `rand`, `randi`, and `randn` will return different values in different MATLAB sessions. It is usually not

desirable to do this more than once per MATLAB session as it may affect the statistical properties of the random numbers MATLAB produces:

```
s = RandStream('mt19937ar','Seed','shuffle');
RandStream.setGlobalStream(s);
```

See Also

RandStream | RandStream.rand | RandStream.randn | RandStream.randi | RandStream.getGlobalStream | RandStream.setGlobalStream | RandStream.list | rng

RandStream.getGlobalStream

Purpose

Current global random number stream

Class

`@RandStream`

Syntax

```
stream = RandStream.getGlobalStream
```

Description

`stream = RandStream.getGlobalStream` returns the global random number stream. The MATLAB functions `rand`, `randi`, and `randn` use the global stream to generate values.

`rand`, `randi`, and `randn` all rely on a stream of uniform pseudorandom numbers known as the *global stream*. `randi` uses one uniform value from the global stream to generate each integer value; `randn` uses one or more uniform values from the global stream to generate each normal value. Note that there are also `rand`, `randi`, and `randn` methods for which you specify a specific random stream from which to draw values.

The `rng` function is a shorter alternative for many common uses of `RandStream.getGlobalStream`.

See Also

`RandStream` | `rng` | `RandStream.setGlobalStream` | `rand` | `randi` | `randn`

Purpose

Set global random number stream

Syntax

```
prevstream = RandStream.setGlobalStream(stream)
```

Description

`prevstream = RandStream.setGlobalStream(stream)` designates the random number stream `stream` as the stream to be used by the `rand`, `randi`, and `randn` functions, and returns the previous global random number stream.

`rand`, `randi`, and `randn` all rely on the same stream of uniform pseudorandom numbers, known as the *global stream*. `randi` uses one uniform value from the global stream to generate each integer value. `randn` uses one or more uniform values from the global stream to generate each normal value. Note that there are also `rand`, `randi`, and `randn` methods for which you specify a specific random stream from which to draw values.

The `rng` function is a shorter alternative for many common uses of `RandStream.setGlobalStream`.

See Also

`RandStream` | `RandStream.getGlobalStream` | `rng` | `rand` | `randi` | `randn`

Purpose

Rank of matrix

Syntax

```
k = rank(A)
k = rank(A,tol)
```

Description

The `rank` function provides an estimate of the number of linearly independent rows or columns of a full matrix.

`k = rank(A)` returns the number of singular values of `A` that are larger than the default tolerance, `max(size(A))*eps(norm(A))`.

`k = rank(A,tol)` returns the number of singular values of `A` that are larger than `tol`.

Tips

Use `sprank` to determine the structural rank of a sparse matrix.

Algorithms

There are a number of ways to compute the rank of a matrix. MATLAB software uses the method based on the singular value decomposition, or SVD. The SVD algorithm is the most time consuming, but also the most reliable.

The `rank` algorithm is

```
s = svd(A);
tol = max(size(A))*eps(max(s));
r = sum(s > tol);
```

See Also

`sprank`

Purpose

Rational fraction approximation

Syntax

```
[N,D] = rat(X)
[N,D] = rat(X,tol)
rat(X)
```

Description

Even though all floating-point numbers are rational numbers, it is sometimes desirable to approximate them by simple rational numbers, which are fractions whose numerator and denominator are small integers. The `rat` function attempts to do this. Rational approximations are generated by truncating continued fraction expansions. The `rats` function calls `rat`, and returns strings.

`[N,D] = rat(X)` returns arrays `N` and `D` so that `N./D` approximates `X` to within the default tolerance, `1.e-6*norm(X(:),1)`.

`[N,D] = rat(X,tol)` returns `N./D` approximating `X` to within `tol`.

`rat(X)`, with no output arguments, simply displays the continued fraction.

Examples

Ordinarily, the statement

```
s = 1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6 + 1/7
```

produces

```
s =
   0.7595
```

However, with

```
format rat
```

or with

```
rats(s)
```

the printed result is

```
s =
   319/420
```

This is a simple rational number. Its denominator is 420, the least common multiple of the denominators of the terms involved in the original expression. Even though the quantity `s` is stored internally as a binary floating-point number, the desired rational form can be reconstructed.

To see how the rational approximation is generated, the statement `rat(s)` produces

```
1 + 1/(-4 + 1/(-6 + 1/(-3 + 1/(-5))))
```

And the statement

```
[n,d] = rat(s)
```

produces

```
n = 319, d = 420
```

The mathematical quantity π is not a rational number, but the MATLAB quantity `pi` that approximates it is a rational number. `pi` is the ratio of a large integer and 2^{52}:

```
14148475504056880/4503599627370496
```

However, this is not a simple rational number. The value printed for `pi` with `format rat`, or with `rats(pi)`, is

```
355/113
```

This approximation was known in Euclid's time. Its decimal representation is

```
3.14159292035398
```

and so it agrees with `pi` to seven significant figures. The statement

```
rat(pi)
```

produces

```
3 + 1/(7 + 1/(16))
```

This shows how the 355/113 was obtained. The less accurate, but more familiar approximation 22/7 is obtained from the first two terms of this continued fraction.

Algorithms

The rat(X) function approximates each element of X by a continued fraction of the form

$$\frac{n}{d} = d_1 + \cfrac{1}{d_2 + \cfrac{1}{\left(d_3 + \ldots + \cfrac{1}{d_k}\right)}}$$

The ds are obtained by repeatedly picking off the integer part and then taking the reciprocal of the fractional part. The accuracy of the approximation increases exponentially with the number of terms and is worst when X = sqrt(2). For x = sqrt(2) , the error with k terms is about 2.68*(.173)^k, so each additional term increases the accuracy by less than one decimal digit. It takes 21 terms to get full floating-point accuracy.

See Also

format | rats

Purpose

Create rubberband box for area selection

Syntax

```
rbbox
rbbox(initialRect)
rbbox(initialRect,fixedPoint)
rbbox(initialRect,fixedPoint,stepSize)
finalRect = rbbox(...)
```

Description

`rbbox` initializes and tracks a rubberband box in the current figure. It sets the initial rectangular size of the box to 0, anchors the box at the figure's `CurrentPoint`, and begins tracking from this point.

`rbbox(initialRect)` specifies the initial location and size of the rubberband box as `[x y width height]`, where `x` and `y` define the lower left corner, and `width` and `height` define the size. `initialRect` is in the units specified by the current figure's `Units` property, and measured from the lower left corner of the figure window. The corner of the box closest to the pointer position follows the pointer until `rbbox` receives a button-up event.

`rbbox(initialRect,fixedPoint)` specifies the corner of the box that remains fixed. All arguments are in the units specified by the current figure's `Units` property, and measured from the lower left corner of the figure window. `fixedPoint` is a two-element vector, `[x y]`. The tracking point is the corner diametrically opposite the anchored corner defined by `fixedPoint`.

`rbbox(initialRect,fixedPoint,stepSize)` specifies how frequently the rubberband box is updated. When the tracking point exceeds `stepSize` figure units, `rbbox` redraws the rubberband box. The default stepsize is `1`.

`finalRect = rbbox(...)` returns a four-element vector, `[x y width height]`, where `x` and `y` are the *x* and *y* components of the lower left corner of the box, and `width` and `height` are the dimensions of the box.

Tips

rbbox is useful for defining and resizing a rectangular region:

- For box definition, initialRect is [x y 0 0], where (x,y) is the figure's CurrentPoint.
- For box resizing, initialRect defines the rectangular region that you resize (e.g., a legend). fixedPoint is the corner diametrically opposite the tracking point.

rbbox returns immediately if a button is not currently pressed. Therefore, you use rbbox with waitforbuttonpress so that the mouse button is down when rbbox is called. rbbox returns when you release the mouse button.

Examples

Assuming the current view is view(2), use the current axes' CurrentPoint property to determine the extent of the rectangle in dataspace units:

```
k = waitforbuttonpress;
point1 = get(gca,'CurrentPoint');    % button down detected
finalRect = rbbox;                   % return figure units
point2 = get(gca,'CurrentPoint');    % button up detected
point1 = point1(1,1:2);              % extract x and y
point2 = point2(1,1:2);
p1 = min(point1,point2);             % calculate locations
offset = abs(point1-point2);         % and dimensions
x = [p1(1) p1(1)+offset(1) p1(1)+offset(1) p1(1) p1(1)];
y = [p1(2) p1(2) p1(2)+offset(2) p1(2)+offset(2) p1(2)];
hold on
axis manual
plot(x,y)                            % redraw in dataspace units
```

See Also

axis | dragrect | waitforbuttonpress

rcond

Purpose

Matrix reciprocal condition number estimate

Syntax

```
c = rcond(A)
```

Description

`c = rcond(A)` returns an estimate for the reciprocal of the condition of `A` in 1-norm. If `A` is well conditioned, `rcond(A)` is near 1.0. If `A` is badly conditioned, `rcond(A)` is near 0.0. Compared to `cond`, `rcond` is a more efficient, but less reliable, method of estimating the condition of a matrix.

See Also

`cond` | `condest` | `norm` | `normest` | `rank` | `svd`

Purpose

Right array division

Syntax

```
x = A./B
x = rdivide(A,B)
```

Description

`x = A./B` divides each element of `A` by the corresponding element of `B`.

- If `A` and `B` are arrays, then they must be the same size.
- If either `A` or `B` is a scalar, then MATLAB expands the scalar value into an appropriately sized array.

`x = rdivide(A,B)` is an alternative way to divide `A` by `B`, but is rarely used. It enables operator overloading for classes.

Input Arguments

A - Numerator

numeric array | sparse numeric array

Numerator, specified as a full or sparse numeric array. If `B` is an integer data type, `A` must be the same integer type or a scalar double.

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

B - Denominator

numeric array | sparse numeric array

Denominator, specified as a full or sparse numeric array. If `A` is an integer data type, `B` must be the same integer type or a scalar double.

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

Output Arguments

x - Solution

numeric array | sparse numeric array

Solution, specified as a sparse or full numeric array. If either A or B are integer data types, then x is that same integer data type.

Examples

Divide Two Numeric Arrays

```
A = [2 4 6 8;3 5 7 9];
B = 10*ones(2,4);
x = A./B

x =

    0.2000    0.4000    0.6000    0.8000
    0.3000    0.5000    0.7000    0.9000
```

Integer Division

MATLAB rounds the results when dividing integer data types.

```
a = int16(10);
b = int16([3 4 6]);
x = a./b

x =

      3      3      2
```

Divide a Scalar by a Numeric Array

```
C = 5;
D = magic(3);
x = C./D

x =

    0.6250    5.0000    0.8333
    1.6667    1.0000    0.7143
    1.2500    0.5556    2.5000
```

Tips

- When dividing integers, use `idivide` for more rounding options.
- MATLAB does not support complex integer division.

See Also

`ldivide | mldivide | mrdivide | Arithmetic Operators \,/ | idivide`

VideoReader.read

Purpose

Read video frame data from file

Syntax

```
video = read(obj)
video = read(obj,index)
video = read(___,'native')
```

Description

`video = read(obj)` reads in all video frames from the file associated with `obj`. The `read` method returns a `H`-by-`W`-by-`B`-by-`F` matrix, `video`, where `H` is the image frame height, `W` is the image frame width, `B` is the number of bands in the image (for example, 3 for RGB), and `F` is the number of frames read.

`video = read(obj,index)` reads only the specified frames. `index` can be a single number or a two-element array representing an index range of the video stream.

`video = read(___,'native')` returns data in the format specified by the `VideoFormat` property, and can include any of the input arguments in the previous syntaxes.

Input Arguments

obj

Name of multimedia object created with `VideoReader`.

index

Frames to read, where the first frame number is 1. Use `Inf` to represent the last frame of the file.

For example:

```
video = read(obj, 1);         % first frame only
video = read(obj, [1 10]);    % first 10 frames
video = read(obj, Inf);       % last frame only
video = read(obj, [50 Inf]);  % frame 50 thru end
```

MATLAB cannot determine the number of frames in a variable frame rate file until you read the last frame. If the requested *index* extends beyond the end of the file, `read` returns either a

warning or an error. For more information, see “Reading Variable Frame Rate Video”.

Default: `[1 Inf]`

Output Arguments

video

Array of data representing video frames, returned in a format dependent on the `VideoFormat` property of `obj`. For most files, the data type and dimensions of `video` are as follows. Note that when the `VideoFormat` property of `obj` is `'Indexed'`, the data type and dimensions of `video` depend on whether you call `read` with the `'native'` argument.

Value of `obj.VideoFormat`	Data Type of `video`	Dimensions of `video`	Description
`'RGB24'`	`uint8`	H-by-W-by-3-by-F	RGB24 image
`'Grayscale'`	`uint8`	H-by-W-by-1-by-F	Grayscale image
`'Indexed'`, without specifying `'native'`	`uint8`	H-by-W-by-3-by-F	RGB24 image
`'Indexed'`, specifying `'native'`	`struct`	1-by-F	MATLAB movie, which is an array of frame structure arrays, each containing the fields `cdata` and `colormap`.

H is the image frame height, W is the image frame width, and F is the number of frames read.

For Motion JPEG 2000 files, the data type and dimensions of video are as follows.

Value of obj.VideoFormat	Data Type of video	Dimensions of video	Description
'Mono8'	uint8	H-by-W-by-1-by-F	Mono image
'Mono8 Signed'	int8	H-by-W-by-1-by-F	Mono signed image
'Mono16'	uint16	H-by-W-by-1-by-F	Mono image
'Mono16 Signed'	int16	H-by-W-by-1-by-F	Mono signed image
'RGB24'	uint8	H-by-W-by-3-by-F	RGB24 image
'RGB24 Signed'	uint8	H-by-W-by-3-by-F	RGB24 signed image
'RGB48'	uint16	H-by-W-by-3-by-F	RGB48 image
'RGB48 Signed'	int16	H-by-W-by-3-by-F	RGB48 signed image

Examples

Read and play back the movie file xylophone.mpg:

```
xyloObj = VideoReader('xylophone.mpg');

nFrames = xyloObj.NumberOfFrames;
vidHeight = xyloObj.Height;
vidWidth = xyloObj.Width;

% Preallocate movie structure.
mov(1:nFrames) = ...
    struct('cdata', zeros(vidHeight, vidWidth, 3, 'uint8'),...
           'colormap', []);
```

```
% Read one frame at a time.
for k = 1 : nFrames
    mov(k).cdata = read(xyloObj, k);
end

% Size a figure based on the video's width and height.
hf = figure;
set(hf, 'position', [150 150 vidWidth vidHeight])

% Play back the movie once at the video's frame rate.
movie(hf, mov, 1, xyloObj.FrameRate);
```

See Also

movie | VideoReader

How To

- "Read Video Files"

Tiff.read

Purpose

Read entire image

Syntax

```
imageData = tiffobj.read()
[Y,Cb,Cr] = tiffobj.read()
```

Description

`imageData = tiffobj.read()` reads the image data from the current image file directory (IFD) in the TIFF file associated with the `Tiff` object, `tiffobj`.

`[Y,Cb,Cr] = tiffobj.read()`reads the YCbCr component data from the current directory in the TIFF file. Depending upon the values of the `YCbCrSubSampling` tag, the size of the `Cb` and `Cr` channels might differ from the `Y` channel.

Examples

Open a `Tiff` object and read data from the TIFF file:

```
t = Tiff('mytif.tif', 'r');
imageData = t.read();
```

See Also

`Tiff.write`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

Purpose

Read data asynchronously from device

Syntax

```
readasync(obj)
readasync(obj,size)
```

Description

`readasync(obj)` initiates an asynchronous read operation on the serial port object, `obj`.

`readasync(obj,size)` asynchronously reads, at most, the number of bytes given by `size`. If `size` is greater than the difference between the `InputBufferSize` property value and the `BytesAvailable` property value, an error is returned.

Tips

Before you can read data, you must connect `obj` to the device with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the device.

You should use `readasync` only when you configure the `ReadAsyncMode` property to `manual`. `readasync` is ignored if used when `ReadAsyncMode` is `continuous`.

The `TransferStatus` property indicates if an asynchronous read or write operation is in progress. You can write data while an asynchronous read is in progress because serial ports have separate read and write pins. You can stop asynchronous read and write operations with the `stopasync` function.

You can monitor the amount of data stored in the input buffer with the `BytesAvailable` property. Additionally, you can use the `BytesAvailableFcn` property to execute a callback function when the terminator or the specified amount of data is read.

Rules for Completing an Asynchronous Read Operation

An asynchronous read operation with `readasync` completes when one of these conditions is met:

- The terminator specified by the `Terminator` property is read.

- The time specified by the `Timeout` property passes.
- The specified number of bytes is read.
- The input buffer is filled (if `size` is not specified).

Because `readasync` checks for the terminator, this function can be slow. To increase speed, you might want to configure `ReadAsyncMode` to `continuous` and continuously return data to the input buffer as soon as it is available from the device.

Examples

This example creates the serial port object `s` on a Windows platform. It connects `s` to a Tektronix TDS 210 oscilloscope, configures `s` to read data asynchronously only if `readasync` is issued, and configures the instrument to return the peak-to-peak value of the signal on channel 1.

```
s = serial('COM1');
fopen(s)
s.ReadAsyncMode = 'manual';
fprintf(s,'Measurement:Meas1:Source CH1')
fprintf(s,'Measurement:Meas1:Type Pk2Pk')
fprintf(s,'Measurement:Meas1:Value?')
```

Begin reading data asynchronously from the instrument using `readasync`. When the read operation is complete, return the data to the MATLAB workspace using `fscanf`.

```
readasync(s)
s.BytesAvailable
ans =
     15
out = fscanf(s)
out =
2.0399999619E0
fclose(s)
```

See Also

`fopen` | `stopasync` | `BytesAvailable` | `BytesAvailableFcn` | `ReadAsyncMode` | `Status` | `TransferStatus`

Purpose

Read data from specified strip

Syntax

```
stripData = tiffobj.readEncodedStrip(stripNumber)
[Y,Cb,Cr] = tiffobj.readEncodedStrip(stripNumber)
```

Description

`stripData = tiffobj.readEncodedStrip(stripNumber)` reads data from the strip specified by `stripNumber`. Strip numbers are one-based numbers.

`[Y,Cb,Cr] = tiffobj.readEncodedStrip(stripNumber)` reads YCbCr component data from the specified strip. The size of the chrominance components `Cb` and `Cr` might differ from the size of the luminance component `Y` depending on the value of the `YCbCrSubSampling` tag.

`readEncodeStrip` clips the last strip, if the strip extends past the `ImageLength` boundary.

Examples

Open a `Tiff` object and read a strip of data. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path.

```
t = Tiff('myfile.tif', 'r');
%
% Check if image is tiled or stipped.
if ~t.isTiled()
  data = t.readEncodedStrip(1);
end
```

References

This method corresponds to the `TIFFReadEncodedStrip` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff.readEncodedTile` | `Tiff.isTiled`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

Tiff.readEncodedTile

Purpose

Read data from specified tile

Syntax

```
tileData = tiffobj.readEncodedTile(tileNumber)
[Y,Cb,Cr] = tiffobj.readEncodedTile(tileNumber)
```

Description

`tileData = tiffobj.readEncodedTile(tileNumber)` reads data from the tile specified by `tileNumber`. Tile numbers are one-based numbers.

`[Y,Cb,Cr] = tiffobj.readEncodedTile(tileNumber)` reads YCbCr component data from the specified tile. The size of the chrominance components `Cb` and `Cr` might differ from the size of the luminance component `Y`, depending on the value of the `YCbCrSubSampling` tag.

`readEncodedTile` clips tiles on the last row or right-most column of an image if the tile extends past the `ImageLength` and `ImageLength` boundaries.

Examples

Open a `Tiff` object and read a tile of data. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path.

```
t = Tiff('myfile.tif', 'r');
%
% Check if image is tiled or stipped.
if t.isTiled()
  data = t.readEncodedTile(1);
end
```

References

This method corresponds to the `TIFFReadEncodedTile` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff.readEncodedStrip` | `Tiff.isTiled`

Tutorials

- “Exporting Image Data and Metadata to TIFF Files”

- "Reading Image Data and Metadata from TIFF Files"

Tiff.readRGBAImage

Purpose

Read image using RGBA interface

Syntax

`[RGB,alpha] = readRGBAImage()`

Description

`[RGB,alpha] = readRGBAImage()` reads an entire image using the RGBA interface. `RGB` consists of an `m`-by-`n`-by-3 colormetric image, where `m` and `n` are the height and width of the tile, respectively. `alpha` is the associated alpha matting. If the image does not have associated alpha matting, then `alpha` is a matrix with all values set to 255 (transparent).

The pixel values may be transformed depending upon the values of the following tags:

`PhotometricInterpretation`

`BitsPerSample`

`SamplesPerPixel`

`Orientation`

`ExtraSamples`

`ColorMap`

Examples

Return all image data as RGB, with associated alpha matting.

```
t = Tiff('example.tif','r');
t.setDirectory(2);
[RGB,A] = t.readRGBAImage();
t.close();
```

References

This method corresponds to the `TIFFReadRGBAImage` function in the LibTIFF C API.

To use this method, you must be familiar with LibTIFF version 4.0.0, as well as the TIFF specification and technical notes. View this documentation at `LibTiff - TIFF Library and Utilities`.

See Also

`Tiff.readRGBAStrip` | `Tiff.readRGBATile` | `Tiff.read`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

Tiff.readRGBAStrip

Purpose

Read strip data using RGBA interface

Syntax

```
[RGB,alpha] = readRGBAStrip(row)
```

Description

`[RGB,alpha] = readRGBAStrip(row)` reads a strip using the RGBA interface. `row` is a one-based number of any row contained by the strip. `RGB` consists of an `m`-by-`n`-by-3 colormetric image, where `m` and `n` are the height and width of the strip, respectively. `alpha` is the associated alpha matting. If the image does not have associated alpha matting, then `alpha` is a matrix with all values set to 255 (transparent).

The strip is clipped if the strip boundary extends past the end of the image.

The pixel values may be transformed depending upon the values of the following tags:

`PhotometricInterpretation`

`BitsPerSample`

`SamplesPerPixel`

`Orientation`

`ExtraSamples`

`ColorMap`

Examples

Open a `Tiff` object and read the strip of data that contains the first row, using the RGBA interface, from the example file, `example.tif`

```
t = Tiff('example.tif','r');
t.setDirectory(2);
[RGB,A] = t.readRGBAStrip(1);
t.close();
```

References

This method corresponds to the `TIFFReadRGBAStrip` function in the LibTIFF C API.

To use this method, you must be familiar with LibTIFF version 4.0.0, as well as the TIFF specification and technical notes. View this documentation at `LibTiff - TIFF Library and Utilities`.

See Also

`Tiff.readRGBATile` | `Tiff.readRGBAImage`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

Tiff.readRGBATile

Purpose

Read tile data using RGBA interface

Syntax

```
[RGB,alpha] = readRGBATile(row,col)
```

Description

`[RGB,alpha] = readRGBATile(row,col)` reads a tile using the RGBA interface. `row` and `col` are the one-based row and column numbers of any pixel in the requested tile. `RGB` consists of an `m`-by-`n`-by-3 colormetric image, where `m` and `n` are the height and width of the tile, respectively. `alpha` is the associated alpha matting. If the image does not have associated alpha matting, then `alpha` is a matrix with all values set to 255 (transparent).

The tile is clipped if the tile boundaries extend past the edges of the image.

The pixel values may be transformed depending upon the values of the following tags:

`PhotometricInterpretation`

`BitsPerSample`

`SamplesPerPixel`

`Orientation`

`ExtraSamples`

`ColorMap`

Examples

Open a `Tiff` object and read the first tile using the RGBA interface.

```
t = Tiff('example.tif','r');
t.setDirectory(1);
[RGB,A] = t.readRGBATile(1,1);
t.close();
```

References

This method corresponds to the `TIFFReadRGBATile` function in the LibTIFF C API.

To use this method, you must be familiar with LibTIFF version 4.0.0, as well as the TIFF specification and technical notes. View this documentation at `LibTiff - TIFF Library and Utilities`.

See Also

`Tiff.readRGBAStrip` | `Tiff.readRGBAImage`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

Remove

Purpose

Convenience function for static .NET System.Delegate Remove method

Syntax

`result = Remove(combinedDelegate,removedDelegate)`

Description

`result = Remove(combinedDelegate,removedDelegate)` removes last instance of the `removedDelegate` delegate from the `combinedDelegate` delegate.

Input Arguments

combinedDelegate

.NET `System.Delegate` object. The combined delegate from which to remove the `removedDelegate` delegate.

removedDelegate

.NET `System.Delegate` object. The delegate to remove from the `combinedDelegate` delegate.

Output Arguments

result

.NET `System.Delegate` object. A new delegate which is the same as the `combinedDelegate` delegate except without the last instance of the `removedDelegate` delegate.

Alternatives

Use the static `Remove` method of the `System.Delegate` class.

See Also

`RemoveAll | Combine`

How To

- "Combine and Remove .NET Delegates"

Related Links

- MSDN System.Delegate.Remove Method reference page

Purpose

Convenience function for static .NET System.Delegate RemoveAll method

Syntax

```
result = RemoveAll(combinedDelegate,removedDelegate)
```

Description

`result = RemoveAll(combinedDelegate,removedDelegate)` removes all instances of `removedDelegate` from `combinedDelegate`.

Input Arguments

combinedDelegate

.NET `System.Delegate` object. The combined delegate from which to remove all instances of the `removedDelegate` delegate.

removedDelegate

.NET `System.Delegate` object. The delegate to remove from the `combinedDelegate` delegate.

Output Arguments

result

.NET `System.Delegate` object. A new delegate which is the same as the `combinedDelegate` delegate except without all instances of the `removedDelegate` delegate.

Alternatives

Use the static `RemoveAll` method of the `System.Delegate` class.

See Also

`Remove` | `Combine`

How To

- "Combine and Remove .NET Delegates"

Related Links

- MSDN System.Delegate.RemoveAll Method reference page

Purpose

Create `timeseries` object

Description

Time series are data vectors sampled over time, in order, often at regular intervals. They are distinguished from randomly sampled data that form the basis of many other data analyses. Time series represent the time-evolution of a dynamic population or process. The linear ordering of time series gives them a distinctive place in data analysis, with a specialized set of techniques. Time series analysis is concerned with:

- Identifying patterns
- Modeling patterns
- Forecasting values

Construction

`ts = timeseries` creates an empty time-series object.

`ts = timeseries(data)` creates the time-series object using the specified `data`.

`ts = timeseries(tsname)` creates the time-series object using the name, `tsname`, for the time-series object. This name can differ from the time-series variable name.

`ts = timeseries(data,time)` creates the time-series object using the specified `data` and `time`.

`ts = timeseries(data,time,quality)` specifies quality in terms of codes defined by `QualityInfo.Code`.

`ts = timeseries(data,Name,Value)` uses one or more `Name, Value` pair arguments.

`ts = timeseries(data,time,Name,Value)` creates the time-series object using the specified (`data`,`time`), time, and `Name, Value` pair arguments.

`ts = timeseries(data,time,quality,Name,Value)` uses the specified `quality`.

Input Arguments

data

The time-series data, which can be an array of samples

tsname

Time-series name specified as a string

Default: ''

time

The time vector.

When time values are date strings, you must specify `Time` as a cell array of date strings. When the time vector contains duplicate values:

- Duplicated values must occupy contiguous elements.
- Time values must not be decreasing.

Interpolating time-series data using methods like `resample` and `synchronize` can produce different results depending on whether the input `timeseries` contains duplicate times.

Default: A time vector that ranges from `0` to `N-1` with a 1-second interval, where `N` is the number of samples.

quality

An integer vector with values`-128` to `127` that specifies the quality in terms of codes defined by `QualityInfo.Code`

When `Quality` is a vector:

- `Quality` must have the same length as the time vector.
- Each `Quality` value applies to the corresponding data sample.

When `Quality` is an array:

- `Quality` must have the same size as the data array.
- Each `Quality` value applies to the corresponding data value of the `ts.data` array.

Name-Value Pair Arguments

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

IsTimeFirst

Specify `'IsTimeFirst',true` when the time vector runs along the first dimension of the data array. Alternatively, if the time vector runs along the last dimension of the data array, specify `'IsTimeFirst',false`.

You can set this property when a 2-D data array is square and, therefore, the dimension that is aligned with time is ambiguous. 3-D and higher dimension data requires `IsTimeFirst` to be `false`; for such data, time steps always lie along the last dimension.

Note In a future release, `IsTimeFirst` will default to `false` for 3-D and higher dimensional data. Setting `IsTimeFirst` to `true` for such data will generate an error.

Default: `true`

Name

Specify `'Name'` followed by a string value if you want to name the `timeseries` object. For example, `'Name','myTSobject'` names the `timeseries` object `'myTSobject'`.

Properties

Data

Time-series data, where each data sample corresponds to a specific time

The data can be a scalar, a vector, or a multidimensional array. Either the first or last dimension of the data must align with `Time`.

By default, `NaN`s represent missing or unspecified data. Set the `TreatNaNasMissing` property to determine how missing data is treated in calculations.

Attributes:

`Dependent`	`true`

DataInfo

Contains fields for storing contextual information about `Data`:

- `Unit` — String that specifies data units
- `Interpolation` — A `tsdata.interpolation` object that specifies the interpolation method for this `timeseries` object.

  Fields of the `tsdata.interpolation` object include:

  - `Fhandle` — Function handle to a user-defined interpolation function
  - `Name` — String that specifies the name of the interpolation method. Predefined methods include `'linear'` and `'zoh'` (zero-order hold). `'linear'` is the default.
- `UserData` — Any user-defined information entered as a string

Events

An array of `tsdata.event` objects that stores event information for this `timeseries` object.

You add events by using the `addevent` method. Fields of the `tsdata.event` object include the following:

- `EventData` — Any user-defined information about the event
- `Name` — String that specifies the name of the event
- `Time` — Time value when this event occurs, specified as a real number or a date string
- `Units` — Time units
- `StartDate` — A reference date specified in MATLAB date-string format. `StartDate` is empty when you have a numerical (non-date-string) time vector.

IsTimeFirst

Logical value (`true` or `false`) specifies whether the first or last dimension of the `Data` array is aligned with the time vector.

You can set this property when the `Data` array is square and it is ambiguous which dimension is aligned with time. By default, the first `Data` dimension that matches the length of the time vector is aligned with the time vector.

When you set this property to:

- `true` — The first dimension of the data array is aligned with the time vector. For example:
  `ts=timeseries(rand(3,3),1:3, 'IsTimeFirst',true);`
- `false` — The last dimension of the data array is aligned with the time vector. For example:
  `ts=timeseries(rand(3,3),1:3, 'IsTimeFirst',false);`

Attributes:

`Dependent`	`true`

Length

Length of the time vector in the `timeseries` object

Attributes:

Dependent	true
SetAccess	'protected'

Name

The `timeseries` object name entered as a string.

This name can differ from the name of the `timeseries` variable in the MATLAB workspace.

Quality

An integer vector or array containing values `-128` to `127` that specify the quality in terms of codes defined by `QualityInfo.Code`.

When `Quality` is a vector, it must have the same length as the time vector. In this case, each `Quality` value applies to a corresponding data sample.

When `Quality` is an array, it must have the same size as the data array. In this case, each `Quality` value applies to the corresponding value of the data array.

Attributes:

Dependent	true

QualityInfo

Provides a lookup table that converts numerical `Quality` codes to readable descriptions.

`QualityInfo` fields include the following:

- `Code` — Integer vector containing values `-128` to `127` that define the “dictionary” of quality codes. You can assign one of these integer values to each `Data` value by using the `Quality` property.
- `Description` — Cell vector of strings, where each element provides a readable description of the associated quality `Code`.

- `UserData` — Stores any additional user-defined information.

Lengths of `Code` and `Description` must match.

Time

Array of time values.

When `TimeInfo.StartDate` is empty, the numerical `Time` values are measured relative to `0` in specified units. When `TimeInfo.StartDate` is defined, the time values are date strings measured relative to the `StartDate` in specified units.

The length of `Time` must be the same as either the first or the last dimension of `Data`.

Attributes:

`Dependent`	`true`

TimeInfo

Uses the following fields for storing contextual information about `Time`:

- `Units` — Time units having any of following values: `'weeks'`, `'days'`, `'hours'`, `'minutes'`, `'seconds'`, `'milliseconds'`, `'microseconds'`, or `'nanoseconds'`
- `Start` — Start time
- `End` — End time (read only)
- `Increment` — Interval between two subsequent time values
- `Length` — Length of the time vector (read only)
- `Format` — String defining the date string display format. See the MATLAB `datestr` function reference page for more information.

- `StartDate` — Date string defining the reference date. See the MATLAB `setabstime` function reference page for more information.
- `UserData` — Stores any additional user-defined information

TreatNaNasMissing

Logical value that specifies how to treat `NaN` values in `Data`:

- `true` — (Default) Treats all `NaN` values as missing data except during statistical calculations.
- `false` — Includes `NaN` values in statistical calculations, in which case `NaN` values are propagated to the result.

UserData

Generic field for data of any class that you want to add to the object.

Default: `[]`

Methods

Time-Series Methods

- Methods to Query and Set Object Properties and Plot the Data on page 4236
- Methods to Manipulate Data and Time on page 4236
- Event Methods on page 4237
- Methods to Arithmetically Combine `timeseries` Objects on page 4238
- Methods to Calculate Descriptive Statistics for a `timeseries` Object on page 4239

Methods to Query and Set Object Properties and Plot the Data

`get`	Query `timeseries` object property values.
`getdatasamplesize`	Return the size of each data sample in a `timeseries` object.
`getqualitydesc`	Return data quality descriptions based on the `Quality` property values assigned to a `timeseries` object.
`plot`	Plot the `timeseries` object.
`set`	Set `timeseries` property values.

Methods to Manipulate Data and Time

`addsample`	Add a data sample to a `timeseries` object.
`append`	Concatenate `timeseries` objects in the time dimension.
`ctranspose`	Transpose a `timeseries` object.
`delsample`	Delete a sample from a `timeseries` object.
`detrend`	Subtract the mean or best-fit line and remove all `NaN`s from time-series data.
`filter`	Shape frequency content of time-series data using a 1-D digital filter.
`getabstime`	Extract a date-string time vector from a `timeseries` object into a cell array.
`getdatasamples`	Extract a subset of data samples from an existing `timeseries` object into an array using a subscripted indexed array.
`getsamples`	Extract a subset of data samples from an existing `timeseries` object into a new `timeseries` object using a subscript indexed array.

(Continued)

getinterpmethod	Get the interpolation method for a timeseries object.
getsampleusingtime	Extract data samples from an existing timeseries object into a new timeseries object based on specified start and end time values.
idealfilter	Apply an ideal pass or notch (noncausal) filter to a timeseries object.
resample	Select or interpolate data in a timeseries object using a new time vector.
setabstime	Set the time values in the time vector as date strings.
setinterpmethod	Set interpolation method for a timeseries object.
setuniformtime	Assign uniform time vector to timeseries object.
synchronize	Synchronize and resample two timeseries objects using a common time vector.
transpose	Transpose a timeseries object.

Event Methods

To construct an event object, use the constructor `tsdata.event`. For an example of defining events for a time-series object, see "Defining Events".

addevent	Add one or more events to a timeseries object.
delevent	Delete one or more events from a timeseries object.

(Continued)

`gettsafteratevent`	Create a new `timeseries` object by extracting the samples from an existing time series that occur after or at a specified event.
`gettsafterevent`	Create a new `timeseries` object by extracting the samples that occur after a specified event from an existing time series.
`gettsatevent`	Create a new `timeseries` object by extracting the samples that occur at the same time as a specified event from an existing time series.
`gettsbeforeatevent`	Create a new `timeseries` object by extracting the samples that occur before or at a specified event from an existing time series.
`gettsbeforeevent`	Create a new `timeseries` object by extracting the samples that occur before a specified event from an existing time series.
`gettsbetweenevents`	Create a new `timeseries` object by extracting the samples that occur between two specified events from an existing time series.

Methods to Arithmetically Combine `timeseries` Objects

`+`	Addition of the corresponding data values of `timeseries` objects.
`-`	Subtraction of the corresponding data values of `timeseries` objects.
`.*`	Element-by-element multiplication of `timeseries` data.

(Continued)

`*`	Matrix-multiply `timeseries` data.
`./`	Right element-by-element division of `timeseries` data.
`/`	Right matrix division of `timeseries` data.
`.\`	Element-by-element left-array divide of `timeseries` data.
`\`	Left matrix division of `timeseries` data.

Methods to Calculate Descriptive Statistics for a `timeseries` Object

`iqr`	Return the interquartile range of `timeseries` data.
`max`	Return the maximum value of `timeseries` data.
`mean`	Return the mean of `timeseries` data.
`median`	Return the median of `timeseries` data.
`min`	Return the minimum of `timeseries` data.
`std`	Return the standard deviation of `timeseries` data.
`sum`	Return the sum of `timeseries` data.
`var`	Return the variance of `timeseries` data.

Definitions

timeseries

The time-series object, called `timeseries`, is a MATLAB variable that contains time-indexed data and properties in a single, coherent structure. For example, in addition to data and time values, you can also use the time-series object to store events, descriptive information about data and time, data quality, and the interpolation method.

Data Sample

A time-series *data sample* consists of one or more values recorded at a specific time. The number of data samples in a time series is the same as the length of the time vector.

For example, suppose that `ts.data` has the size 3-by-4-by-5 and the time vector has the length 5. Then, the number of samples is `5` and the total number of data values is `3 x 4 x 5 = 60`.

Time Vector

A time vector of a `timeseries` object can be either numerical (`double`) values or valid MATLAB date strings.

When the `timeseries` `TimeInfo.StartDate` property is empty, the numerical `time` values are measured relative to 0 (or another numerical value) in specified units. In this case, the time vector is described as *relative* (that is, it contains time values that are not associated with a specific start date).

When `TimeInfo.StartDate` is nonempty, the time values are date strings measured relative to `StartDate` in specified units. In this case, the time vector is described as *absolute* (that is, it contains time values that are associated with a specific calendar date).

MATLAB supports the following date-string formats for time-series applications.

Date-String Format	Usage Example
`dd-mmm-yyyy HH:MM:SS`	`01-Mar-2000 15:45:17`
`dd-mmm-yyyy`	`01-Mar-2000`
`mm/dd/yy`	`03/01/00`
`mm/dd`	`03/01`
`HH:MM:SS`	`15:45:17`
`HH:MM:SS PM`	`3:45:17 PM`
`HH:MM`	`15:45`

Date-String Format	Usage Example
HH:MM PM	3:45 PM
mmm.dd,yyyy HH:MM:SS	Mar.01,2000 15:45:17
mmm.dd,yyyy	Mar.01,2000
mm/dd/yyyy	03/01/2000

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Create a `timeseries` object called `'LaunchData'` that contains four data sets, each stored as a column of length `5` and using the default time vector:

```
b = timeseries(rand(5, 4),'Name','LaunchData')
```

Create a `timeseries` object containing a single data set of length `5` and a time vector starting at `1` and ending at `5`:

```
b = timeseries(rand(5,1),[1 2 3 4 5])
```

Create a `timeseries` object called `'FinancialData'` containing five data points at a single time point:

```
b = timeseries(rand(1,5),1,'Name','FinancialData')
```

Alternatives

Use the “Time Series Tools”.

See Also

`tscollection` | `tsdata.event`

How To

- “MATLAB Objects”

Purpose

Add data sample to `timeseries` object

Syntax

```
ts1 = addsample(ts, s)
ts1 = addsample(ts, 'Data', data-value,
'Time',time-value,...,
   Name,Value)
```

Description

`ts1 = addsample(ts, s)` adds one or more new data samples stored in a structure `s` to the `timeseries` object `ts`.

`ts1 = addsample(ts, 'Data', data-value, 'Time',time-value,..., Name,Value)` adds one or more data samples to the `timeseries` object `ts` along with additional options specified by one or more `Name,Value` pair arguments.

Tips

- If `N` is the number of data samples, you can get the sample size of each time with `SampleSize = getsamplesize(ts)`.

  When `ts.IsTimeFirst` is `true`, the size of the data is `N`-by-`SampleSize`. When `ts.IsTimeFirst` is `false`, the size of the data is `SampleSize`-by-`N`.

Input Arguments

s

A structure that you must define before passing as an argument to `addsample`. It consists of the following optional fields:

- `s.data`
- `s.time`
- `s.quality`
- `s.overwriteflag`

data-value

A numeric data value.

time-value

A valid time vector.

Name-Value Pair Arguments

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

'Quality'

Array of data quality codes.

Default: `[]`

'OverwriteFlag'

Logical value that controls whether to overwrite a data sample at the same time with the new sample you are adding to your `timeseries` object. When set to `true`, the new sample overwrites the old sample at the same time.

Default: `false`

Output Arguments

ts1

The `timeseries` object that results when you add the specified samples to the original `timeseries` object.

Definitions

data sample

One or more values recorded at a specific time. The number of data samples in a time series is the same as the length of the time vector.

Examples

Add a data value of `420` at time `3`:

```
ts = ts.addsample('Time',3,'Data',420);
```

Add a data value of 420 at time 3 and specify quality code 1 for this data value. Set the OverwriteFlag to overwrite an existing value at time 3.

```
ts = ts.addsample('Data',3.2,'Quality',1,'OverwriteFlag',...
        true,'Time',3);
```

See Also

timeseries | delsample | getdatasamples

timeseries.append

Purpose

Concatenate time series objects in time dimension

Syntax

```
ts = append(ts1,ts2, ... tsn)
```

Description

`ts = append(ts1,ts2, ... tsn)` creates a new `timeseries` object by concatenating `timeseries` `ts1`, `ts2`, and so on, along the time dimension.

Tips

- A single overlapping time between each input time series is valid, as long as the overlapping samples are identical.
- The time vectors must not overlap by a nonzero amount. That is, the last time in `ts1` must be earlier than or equal to the first time in `ts2`.
- The sample size of the time series must be the same.

Input Arguments

ts1

The first `timeseries` object that you want to append.

ts2

The second `timeseries` object that you want to append.

tsn

The nth `timeseries` object that you want to append.

Output Arguments

ts

The `timeseries` object that results from appending the input `timeseries` objects.

Examples

After creating `timeseries` objects, `ts1` and `ts2`, append them:

```
ts1 = timeseries(rand(5,1),[1 2 3 4 5]);
ts2 = timeseries(rand(5,1),[6 7 8 9 10]);
ts3 = append(ts1, ts2)
```

See Also

`timeseries`

Purpose

Transpose `timeseries` object

Syntax

```
ts1 = ctranspose(ts)
```

Description

`ts1 = ctranspose(ts)` returns a new `timeseries` object `ts1` with the `IsTimeFirst` value set to opposite of what it is for `ts`. For example, if `ts` has the first data dimension aligned with the time vector, `ts1` has the last data dimension aligned with the time vector as a result of this operation.

Tips

- The overloaded `ctranspose` method for `timeseries` objects does not transpose the data. Instead, this method changes whether the first or the last dimension of the data aligns with the time vector. To transpose the data, you must transpose the `Data` property of the `timeseries` object. For example, you can use the syntax `ctranspose(ts.Data)` or `(ts.Data)'`. The `Data` property value must be a 2-D array.
- Consider a `timeseries` object with 10 samples with the property `IsTimeFirst = True`. When you transpose this object, the data size changes from 10-by-1 to 1-by-1-by-10. Note that the first dimension of the `Data` property is shown explicitly.

  The following table summarizes the size for Data property of the `timeseries` object (up to three dimensions) before and after transposing.

  Data Size Before and After Transposing

Size of Original Data	Size of Transposed Data
N-by-1	1-by-1-by-N
N-by-M	M-by-1-by-N
N-by-M-by-L	M-by-L-by-N

timeseries.ctranspose

Input Arguments

ts

The `timeseries` object you want to transpose.

Output Arguments

ts1

The transposed `timeseries` object.

Examples

Suppose that a `timeseries` object `ts` has `ts.data` size 10-by-3-by-2 and its time vector has a length of 10. The `IsTimeFirst` property of `ts` is set to `true`, which means that the first dimension of the data is aligned with the time vector. `ctranspose(ts)` modifies `ts`, such that the last dimension of the data is now aligned with the time vector. This permutes the data, such that the size of `ts.Data` becomes 3-by-2-by-10.

See Also

`timeseries` | `transpose`

Purpose

Remove sample from `timeseries` object

Syntax

```
ts1 = delsample(ts,Name,Value)
```

Description

`ts1 = delsample(ts,Name,Value)` deletes samples from the `timeseries` object `ts` based on the specified `Name,Value` pair arguments.

Input Arguments

ts

A `timeseries` object.

Name-Value Pair Arguments

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

Index

The indices of the time vector that correspond to the samples you want to delete.

Value

The time values that correspond to the samples you want to delete.

Output Arguments

ts1

The `timeseries` object that results from removing the specified samples.

Examples

Create a `timeseries` object, and then remove samples:

```
ts = timeseries(rand(5,1),[10 20 30 40 50]);

% Remove data sample at time index 1

ts1 = delsample(ts,'Index', 1)
```

```
% Remove data sample at time value 20:

ts2 = delsample(ts,'Value', [20])
```

set | timeseries

timeseries.detrend

Purpose

Subtract mean or best-fit line and all `NaN`s from `timeseries` object

Syntax

```
ts1 = detrend(ts, method)
ts1 = detrend(ts, method, index)
```

Description

`ts1 = detrend(ts, method)` subtracts either a mean or a best-fit line from time-series data, using the specified `method`. Usually for FFT processing.

`ts1 = detrend(ts, method, index)` uses the optional `index` to specify the columns or rows to detrend.

Tips

- You cannot apply `detrend` to`timeseries` data with more than two dimensions.

Input Arguments

ts

The `timeseries` object from which you want to subtract the mean or best-fit line and all `NaN`s.

method

A string that specifies one of the following detrend methods:

- `'constant'` — Subtracts the mean.
- `'linear'` — Subtracts the best-fit line.

index

An integer array that specifies the columns or rows to detrend when `ts.IsTimeFirst` is true.

Output Arguments

ts1

The `timeseries` object resulting from detrending the input `timeseries` object.

See Also

`timeseries`

timeseries.filter

Purpose

Shape frequency content of time-series

Syntax

```
ts1 = filter(ts, numerator, denominator)
ts1=filter(ts, numerator, denominator, index)
```

Description

`ts1 = filter(ts, numerator, denominator)` applies the transfer function filter $b(z^{-1})/a(z^{-1})$ to the data in the `timeseries` object `ts`. `b` and `a` are the coefficient arrays of the transfer function numerator and denominator, respectively.

`ts1=filter(ts, numerator, denominator, index)` uses the optional `index` integer array to specify either the columns or rows to filter, depending on the value of `ts.IsTimeFirst`.

Tips

- The time-series data must be uniformly sampled to use this filter.
- The following function

  ```
  y = filter(b,a,x)
  ```

  creates filtered data `y` by processing the data in vector `x` with the filter described by vectors `a` and `b`.
- The `filter` function is a general tapped delay-line filter, described by the difference equation:

  $$a(1)y(n) = b(1)x(n) + b(2)x(n-1) + \ldots + b(nb)x(n-nb+1) - a(2)y(n-1) - \ldots - a(N_a)y(n-N_b+1).$$

  Here, n is the index of the current sample, N_a is the order of the polynomial described by vector `a`, and N_b is the order of the polynomial described by vector `b`. The output $y(n)$ is a linear combination of current and previous inputs, $x(n)$ $x(n-1)$..., and previous outputs, $y(n-1)$ $y(n-2)$... .
- You use the discrete filter to shape the data by applying a transfer function to the input signal.

Depending on your objectives, the transfer function you choose might alter both the amplitude and the phase of the variations in the data at different frequencies to produce either a smoother or a rougher output.

- In digital signal processing (DSP), it is customary to write transfer functions as rational expressions in z^{-1} and to order the numerator and denominator terms in ascending powers of z^{-1}.

  Taking the z-transform of the difference equation

  $$a(1)y(n) = b(1)x(n) + b(2)x(n-1) + \ldots + b(nb)x(n-nb+1) - a(2)y(n-1) - \ldots - a(na)y(na+1),$$

  results in the transfer function

  $$Y(z) = H(z^{-1})X(z) = \frac{b(1) + b(2)z^{-1} + \ldots + b(nb)z^{-nb+1}}{a(1) + a(2)z^{-1} + \ldots + a(na)z^{-na+1}} X(z),$$

  where $Y(z)$ is the z-transform of the filtered output $y(n)$. The coefficients b and a are unchanged by the z-transform.

Input Arguments

ts

The first `timeseries` object for which you want to shape the frequency content.

numerator

The coefficient array of the transfer function numerator.

denominator

The coefficient array of the transfer function denominator.

index

An integer array that specifies the columns or rows to filter when `ts.IsTimeFirst` is true.

Output Arguments

ts1

The `timeseries` object that results from filtering the input `timeseries` object.

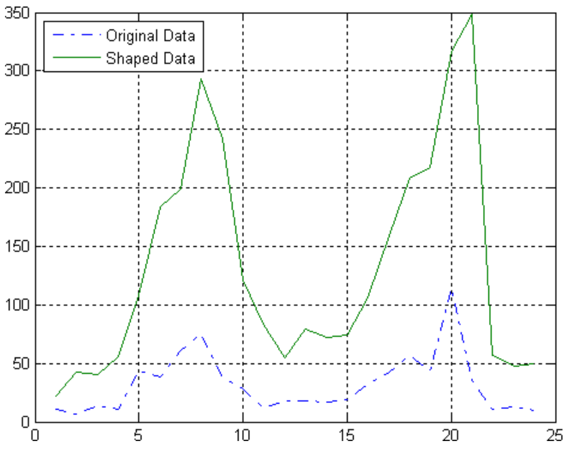
Examples

This example applies the following transfer function to the data in `count.dat`:

$$H(z^{-1}) = \frac{b(z^{-1})}{a(z^{-1})} = \frac{2+3z^{-1}}{1+0.2z^{-1}}$$

```
% Load the matrix count into the workspace:

load count.dat;

% Create a time-series object based on this matrix:

count1=timeseries(count(:,1),[1:24]);

% Enter the coefficients of the denominator ordered in
% ascending powers of z-1 to
%  represent 1 + 0.2x-1:

a = [1 0.2];

% Enter the coefficients of the numerator to represent 2 - 3z-1:

b = [2 3];

% Call the filter method:

filter_count = filter(count1, b, a)

% Compare the original data and the shaped data with an
% overlaid plot of the two curves:

plot(count1,'-.'), grid on, hold on
```

```
plot(filter_count,'-')
legend('Original Data','Shaped Data',2)
```



See Also

timeseries | idealfilter

timeseries.get

Purpose

Query `timeseries` object property values

Syntax

```
get(ts)
value = get(ts, PropertyName)
```

Description

`get(ts)` displays all properties and values of the `timeseries` object, `ts`.

`value = get(ts, PropertyName)` returns the property value for the specified `timeseries` object. The following syntax is equivalent:

```
value = ts.PropertyName
```

Input Arguments

ts

A `timeseries` object.

PropertyName

String specifying the name of a `timeseries` property. For a list of `timeseries` properties, see `timeseries`.

Output Arguments

value

String containing the value associated with the specified property.

Examples

Create a `timeseries` object, and then get the name. This example gets the name three different ways:

```
ts1 = timeseries(rand(5,1),[1 2 3 4 5], 'Name', 'MyTimeseries');
get(ts1)
get(ts1, 'Name')
value = ts1.Name
```

See Also

`timeseries` | `set`

Purpose

Extract date-string time vector into cell array

Syntax

```
getabstime(ts)
```

Description

`getabstime(ts)` extracts the time vector from the `timeseries` object `ts` as a cell array of date strings.

Tips

- To define the time vector relative to a calendar date, set the `TimeInfo.StartDate` property of the `timeseries` object. When the `TimeInfo.StartDate` format is a valid `datestr` format, the output strings from `getabstime` have the same format.

Input Arguments

ts

The `timeseries` object from which you want to extract the time vector.

Examples

The following example extracts a time vector as a cell array of date strings from a `timeseries` object.

First, create a `timeseries` object.

```
ts = timeseries([3 6 8 0 10]);
```

The default time vector for `ts` is `[0 1 2 3 4]`, which starts at 0 and increases in 1-second increments. The length of the time vector is equal to the length of the data.

Next, set the `StartDate` property.

```
ts.TimeInfo.StartDate = '10/27/2005 07:05:36';
```

Extract the time vector.

```
getabstime(ts)
```

MATLAB returns:

```
'27-Oct-2005 07:05:36'
```

```
'27-Oct-2005 07:05:37'
'27-Oct-2005 07:05:38'
'27-Oct-2005 07:05:39'
'27-Oct-2005 07:05:40'
```

Change the date-string format of the time vector, and then extract the time vector with the new date-string format:

```
ts.TimeInfo.Format = 'mm/dd/yy';
getabstime(ts)
```

MATLAB returns:

```
'10/27/05'
'10/27/05'
'10/27/05'
'10/27/05'
'10/27/05'
```

See Also

timeseries | setabstime | datestr

Purpose

Returns subset of time series samples using subscripted index array

Syntax

```
datasamples = getdatasamples(ts, i)
```

Description

`datasamples = getdatasamples(ts, i)` returns an array corresponding to the samples indicated by the array, `i`.

Input Arguments

ts

The `timeseries` object from which you want to exact samples.

i

An array of linear indices or logical values that specifies the time value or values for which you want to extract the corresponding samples.

Output Arguments

datasamples

The array that results from extracting the samples corresponding to the time value or values, `ts.time(i)`.

Examples

After creating a `timeseries` object, `ts`, extract the second and third data samples into an array:

```
ts = timeseries(rand(5,1),[1 2 3 4 5]);
samples = getdatasamples(ts, [2 3])
```

See Also

`timeseries` | `getsamples` | `resample`

timeseries.getdatasamplesize

Purpose

Size of data sample in `timeseries` object

Syntax

```
getdatasamplesize(ts)
```

Description

`getdatasamplesize(ts)` returns the size of each data sample in a `timeseries` object.

Input Arguments

ts

String specifying the name of a `timeseries` object.

Definitions

data sample

One or more scalar values recorded at a specific time. The number of data samples is the same as the length of the time vector.

Examples

After loading data and creating a `timeseries` object, get the size of a data sample:

```
% Load a 24-by-3 data array:
load count.dat

% Create a timeseries object with 24 time values:
count_ts = timeseries(count,[1:24],'Name','VehicleCount')

% Get the size of the data sample for this timeseries object:
getdatasamplesize(count_ts)
```

MATLAB returns the following, which indicates that the size of each data sample in `count_ts` is 1-by-3. In other words, MATLAB stores each data sample as a row with three values.

```
ans =

     1     3
```

See Also

`timeseries` | `set`

Purpose

Interpolation method for `timeseries` object

Syntax

```
getinterpmethod(ts)
```

Description

`getinterpmethod(ts)` returns the interpolation method that the `timeseries` object `ts`, uses as a string.

Input Arguments

ts

The `timeseries` object from which you want to extract the interpolation method.

Definitions

interpolation method

Predefined interpolation methods are zero-order hold, `zoh`, and linear interpolation, `linear`. Linear interpolation is the default.

Examples

Create a `timeseries` object, and then get its interpolation method:

```
ts = timeseries(rand(5));
getinterpmethod(ts)
```

MATLAB returns:

```
linear
```

See Also

`timeseries` | `setinterpmethod`

timeseries.getqualitydesc

Purpose

Data quality descriptions

Syntax

```
getqualitydesc(ts)
```

Description

`getqualitydesc(ts)` returns a cell array of data quality descriptions based on the `Quality` values you assigned to a `timeseries` object, `ts`.

Input Arguments

ts

A `timeseries` object.

Examples

Create a `timeseries` object, and then get the data quality description strings for `ts`:

```
% Create a timeseries object, ts, with Data,
Time, and Quality
% values, respectively:

ts = timeseries([3; 4.2; 5; 6.1; 8], 1:5, [1; 0; 1; 0; 1]);

% Set the QualityInfo property, including Code and Description:

ts.QualityInfo.Code = [0 1];
ts.QualityInfo.Description = {'good' 'bad'};

% Get the data quality description strings for ts:
getqualitydesc(ts)
```

MATLAB returns:

```
ans =

    'bad'
    'good'
    'bad'
    'good'
    'bad'
```

See Also

`timeseries`

timeseries.getsamples

Purpose

Subset of time series samples using subscripted index array

Syntax

```
ts1 = getsamples(ts, i)
```

Description

`ts1 = getsamples(ts, i)` returns a new `timeseries` object by extracting samples from `timeseries` `ts` corresponding to the time or times indicated by the subscripted index array, `i`.

Input Arguments

ts

The `timeseries` object from which you want to exact samples.

i

A subscripted index array that specifies the time value or values for which you want to extract the corresponding samples.

Output Arguments

ts1

The `timeseries` object that results from extracting the samples corresponding to the time value or values `ts.time(i)` .

Examples

After creating a `timeseries` object, `ts`, extract the data samples at times 2 and 3 into a new `timeseries` object, `ts1`:

```
ts = timeseries(rand(5,1),[1 2 3 4 5]);
ts1 = getdatasamples(ts, ts.time([2 3]))
```

See Also

`timeseries` | `getdatasamples` | `resample`

Purpose

Extract data samples into new `timeseries` object

Syntax

```
ts1 = getsampleusingtime(ts, time)
ts1 = getsampleusingtime(ts, starttime, endtime)
```

Description

`ts1 = getsampleusingtime(ts, time)` returns a new `timeseries` object, `ts1`, with a single sample corresponding to specified `time` in `ts`.

`ts1 = getsampleusingtime(ts, starttime, endtime)` returns a new `timeseries` object, `ts1`, with samples between the times `starttime` and `endtime` in `ts`.

Tips

- If the time vector in `ts` is not relative to a calendar date, then `starttime` and `endtime` must be numeric.
- If the time vector in `ts` is relative to a calendar date, then `starttime` and `endtime` values must be dates—either strings or `datenum` values.

Input Arguments

ts

The `timeseries` object from which you want to extract data samples.

time

The time corresponding to data sample you want to extract.

starttime

The time corresponding to the first data sample you want to extract.

endtime

The time corresponding to the last data sample you want to extract.

Output Arguments

ts1

A `timeseries` object that contains the subset of data samples from the original `timeseries` object.

See Also

`timeseries`

timeseries.idealfilter

Purpose

Apply ideal (noncausal) filter to `timeseries` object

Syntax

```
ts1 = idealfilter(ts, interval, filtertype)
ts1 = idealfilter(ts, interval, filtertype, index)
```

Description

`ts1 = idealfilter(ts, interval, filtertype)` applies an ideal filter of `filtertype` to one or more frequency intervals that `interval` specifies for the `timeseries` object, `ts`.

`ts1 = idealfilter(ts, interval, filtertype, index)` applies an ideal filter and uses the optional `index` integer array to specify the columns or rows to filter.

Tips

- Ideal filters require data to have a mean of zero and prepare the data by subtracting its mean. You can restore the filtered signal amplitude by adding the mean of the input data to the filter output values.
- Use the ideal *notch* filter when you want to remove variations in a specific frequency range. Alternatively, use the ideal *pass* filter to allow only the variations in a specific frequency range.
- If the time-series data is sampled nonuniformly, filtering resamples this data on a uniform time vector.
- All `NaN`s in the time series are interpolated before filtering, using the interpolation method you assigned to the `timeseries` object.

Input Arguments

ts

The `timeseries` object to which you want to apply an ideal filter.

interval

The frequency interval (specified in cycles per time unit) at which you want the ideal filter applied. To specify several frequency intervals, use an n-by-2 array of start and end frequencies, where `n` represents the number of intervals.

filtertype

A string specifying the type of filter you want to apply, either `pass` or `notch`.

index

An integer array that specifies the columns or rows to filter when `ts.IsTimeFirst` is true.

Output Arguments

ts1

The `timeseries` object that results when you apply an ideal filter to the original `timeseries` object.

Definitions

ideal filter

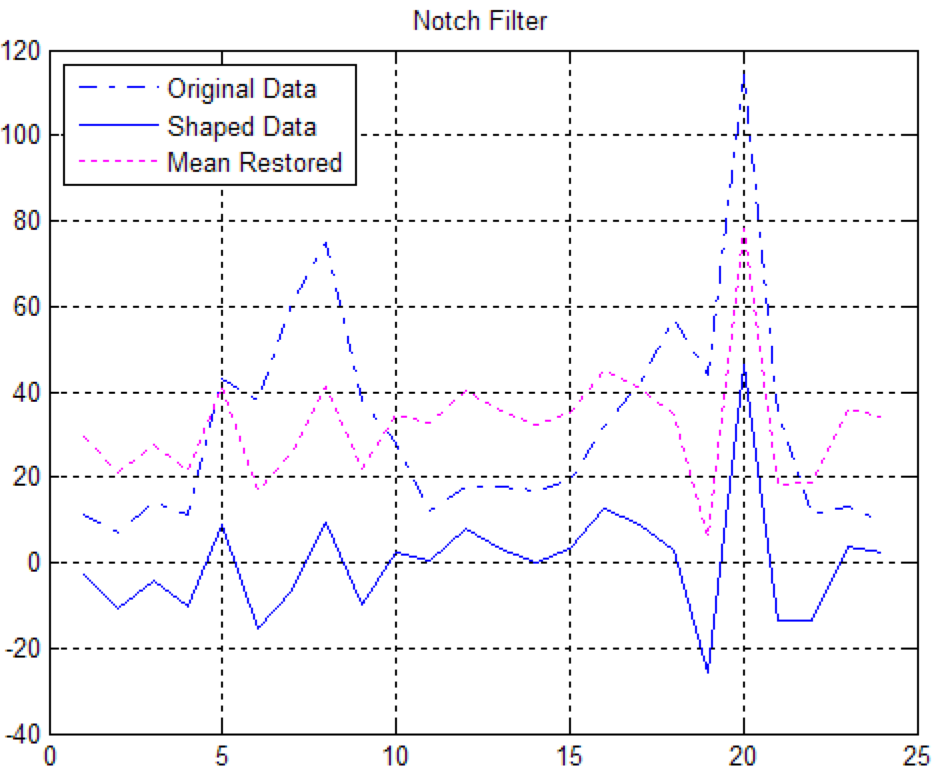
Filters are *ideal* in the sense that they are *not realizable*. An ideal filter is noncausal and the ends of the filter amplitude are perfectly flat in the frequency domain.

Examples

These examples first apply an ideal notch filter to the data in `count.dat`. Then, they apply a pass filter to the data:

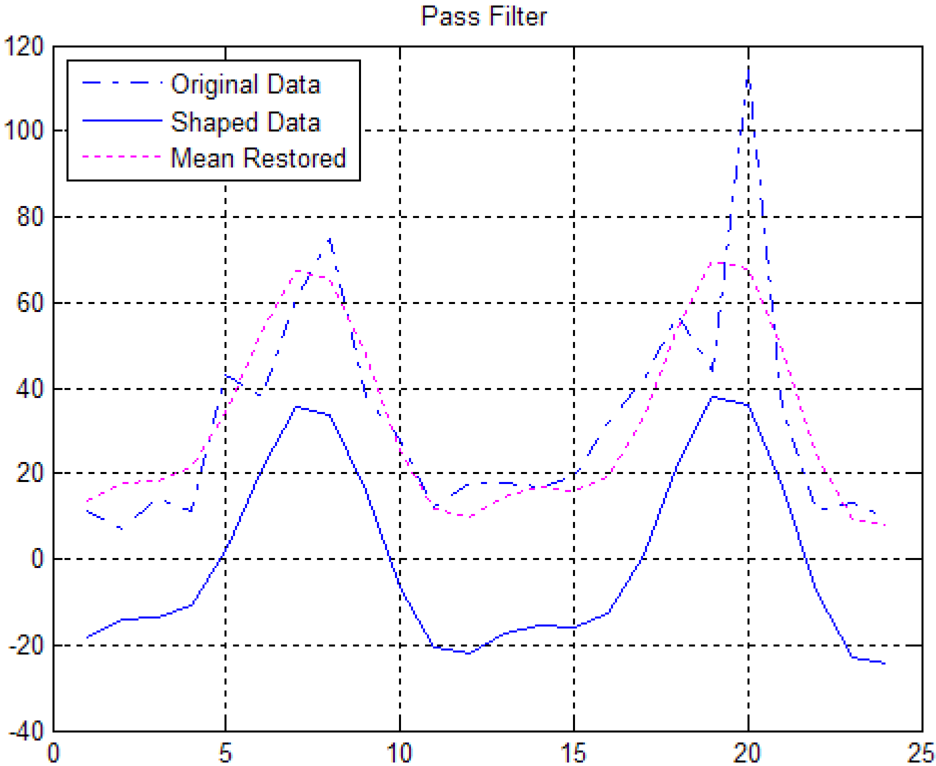
```
% Load the count matrix into the workspace:

load count.dat;

% Create a timeseries object from column one of this matrix.
% Specify a time vector that ranges from 1 to 24 s in 1-s intervals.

count1=timeseries(count(:,1),1:24);

% Obtain the mean of the data:

countmean = mean(count1);

% Enter the frequency interval, in hertz, for filtering the data:
interval=[0.08 0.2];
```

```
% Invoke an ideal notch filter:
idealfilter_countn = idealfilter(count1,interval,'notch')

% Compare the original data and the shaped data on a line plot:

plot(count1,'-.'), grid on, hold on
plot(idealfilter_countn,'-')

% Restore the mean to the filtered data and show it on the line plot,
% adding a legend and a title:

countn_restored = idealfilter_countn + countmean;
plot(countn_restored,':m');
title('Notch Filter')
legend('Original Data','Shaped Data','Mean Restored',...
       'Location','NorthWest')
```




```
% Close the Figure window:
close
% Then, repeat the process using a pass rather than a notch filter:
figure
plot(count1,'-.'), grid on, hold on
idealfilter_countp = idealfilter(count1,interval,'pass');
plot(idealfilter_countp,'-')
countp_restored = idealfilter_countp + countmean;
plot(countp_restored,':m');
title('Pass Filter')
legend('Original Data','Shaped Data','Mean Restored',...
```

```
'Location','NorthWest')
```



See Also

`timeseries` | `filter`

timeseries.iqr

Purpose

Interquartile range of `timeseries` data

Syntax

```
ts_iqr = iqr(ts)
iqr(ts, Name, Value)
```

Description

`ts_iqr = iqr(ts)` returns the interquartile range of `ts.Data`.

`iqr(ts, Name, Value)` reruns the interquatrile range of `ts.Data` with the specified Name, Value pairs.

Input Arguments

ts

The `timeseries` object for which you want the interquartile range of `timeseries` data.

Name-Value Pair Arguments

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

'MissingData'

A string specifying one of two possible values, `remove` or `interpolate`, indicating how to treat missing data during the calculation.

Default: `remove`

'Quality'

A vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).

Output Arguments

ts_iqr

The interquartile range of `ts.Data`, as follows:

- When `ts.Data` is a vector, `ts_iqr` is the difference between the 75th and the 25th percentiles of the `ts.Data` values.
- When `ts.Data` is a matrix, and `IsTimeFirst` is `true`, and the first dimension of `ts` is aligned with time, then `ts_iqr` is a row vector containing the interquartile range of each column of `ts.Data`.

When `ts.Data` is an N-dimensional array, `iqr` always operates along the first nonsingleton dimension of `ts.Data`.

Examples

Create a time series with a missing value, represented by `NaN`, and then calculate the interquartile range of `ts.Data` after removing the missing value from the calculation:

```
ts = timeseries([3.0 NaN 5 6.1 8], 1:5);
iqr(ts,'MissingData','remove')
```

MATLAB returns:

```
    3.0500
```

See Also

max | mean | median | min | std | sum | timeseries | var

timeseries.max

Purpose

Maximum value of `timeseries` data

Syntax

```
ts_max = max(ts)
ts_max = max(ts,Name,Value)
```

Description

`ts_max = max(ts)` returns the maximum value in the `timeseries` data.

`ts_max = max(ts,Name,Value)` returns the maximum value in the `timeseries` data with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

ts

The `timeseries` object for which you want to determine the maximum data value.

Name-Value Pair Arguments

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

'MissingData'

A string specifying one of two possible values, `remove` or `interpolate`, indicating how to treat missing data during the calculation.

Default: `remove`

'Quality'

A vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).

Output Arguments

ts_max

The maximum data value in the specified `timeseries` object, as follows:

- When `ts.Data` is a vector, `ts_max` is the maximum value of `ts.Data` values.
- When `ts.Data` is a matrix, `IsTimeFirst` is `true`, and the first dimension of `ts` is aligned with time, then `ts_max` is a row vector containing the maximum value of each column of `ts.Data`.

When `ts.Data` is an N-dimensional array, `max` always operates along the first nonsingleton dimension of `ts.Data`.

Examples

The following example illustrates how to find the maximum values in multivariate time-series data:

```
% Load a 24-by-3 data array:
load count.dat

% Create a timeseries object with 24 time values:
count_ts = timeseries(count,[1:24],'Name','CountPerSecond')

% Find the maximum in each data column for this timeseries object:

max(count_ts)
```

MATLAB returns:

```
114   145   257
```

See Also

`iqr` | `mean` | `median` | `min` | `std` | `sum` | `timeseries` | `var`

timeseries.mean

Purpose

Mean value of `timeseries` data

Syntax

```
ts_mn = mean(ts)
ts_mn = mean(ts,Name,Value)
```

Description

`ts_mn = mean(ts)` returns the mean value of `ts.Data`.

`ts_mn = mean(ts,Name,Value)` returns the mean value of `ts.Data` with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

ts

The `timeseries` object for which you want the mean value of data.

Name-Value Pair Arguments

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

'MissingData'

A string specifying one of two possible values, `remove` or `interpolate`, indicating how to treat missing data during the calculation.

Default: `remove`

'Quality'

A vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).

'Weighting'

A string specifying one of two possible values, `none` or `time`. When you specify `time`, larger time values correspond to larger weights.

Output Arguments

ts_mn

The mean value of `ts.Data`, as follows:

- When `ts.Data` is a vector, `ts_mn` is the mean value of `ts.Data` values.
- When `ts.Data` is a matrix, and `IsTimeFirst` is `true` and the first dimension of `ts` is aligned with time, then `ts_mn` is a row vector containing the mean value of each column of `ts.Data`.

When `ts.Data` is an N-dimensional array, `mean` always operates along the first nonsingleton dimension of `ts.Data`.

Examples

Find the mean values in multivariate time-series data:

```
% Load a 24-by-3 data array:

load count.dat

%  Create a timeseries object with 24 time values:

count_ts = timeseries(count,[1:24],'Name','CountPerSecond')

% Find the mean of each data column for this timeseries object:

mean(count_ts)
```

MATLAB returns:

```
32.0000   46.5417   65.5833
```

Algorithms

MATLAB determines weighting by:

1. Attaching a weighting to each time value, depending on its order, as follows:
   - First time point — The duration of the first time interval `(t(2) - t(1))`.
   - Time point that is neither the first nor last time point — The duration between the midpoint of the previous time interval to the midpoint of the subsequent time interval `((t(k + 1) - t(k))/2 + (t(k) - t(k - 1))/2)`.
   - Last time point — The duration of the last time interval `((t(end) - t(end - 1))`.
2. Normalizing the weighting for each time by dividing each weighting by the mean of all weightings.

   Note If the `timeseries` object is uniformly sampled, then the normalized weighting for each time is 1.0. Therefore, time weighting has no effect.

3. Multiplying the data for each time by its normalized weighting.

See Also

`timeseries` | `timeseries.iqr` | `timeseries.max` | `timeseries.median` | `timeseries.min` | `timeseries.std` | `timeseries.sum` | `timeseries.var`

Purpose

Median value of `timeseries` data

Syntax

```
ts_med = median(ts)
ts_med = method(ts,Name,Value)
```

Description

`ts_med = median(ts)` returns the median value of `ts.Data`.

`ts_med = method(ts,Name,Value)` returns the median value of `ts.Data` with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

ts

The `timeseries` object for which you want the median data value.

Name-Value Pair Arguments

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

'MissingData'

A string specifying one of two possible values, `remove` or `interpolate`, indicating how to treat missing data during the calculation.

Default: `remove`

'Quality'

A vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).

'Weighting'

A string specifying one of two possible values, none or time. When you specify time, larger time values correspond to larger weights.

Output Arguments

ts_med

The median value of ts.Data, as follows:

- When ts.Data is a vector, ts_med is the mean value of ts.Data values.
- When ts.Data is a matrix, and IsTimeFirst is true, and the first dimension of ts is aligned with time, then ts_med is a row vector containing the median value of each column of ts.Data.

When ts.Data is an N-dimensional array, median always operates along the first nonsingleton dimension of ts.Data.

Examples

The following example finds the median values in multivariate time-series data. MATLAB finds the median independently for each data column in the timeseries object:

```
% Load a 24-by-3 data array:

load count.dat

% Create a timeseries object with 24 time values:

count_ts = timeseries(count,[1:24],'Name','CountPerSecond');

% Find the median of each data column for this timeseries object:

median(count_ts)
```

MATLAB returns:

```
23.5000   36.0000   39.0000
```

Algorithms

MATLAB determines weighting by:

1. Attaching a weighting to each time value, depending on its order, as follows:
   - First time point — The duration of the first time interval `(t(2) - t(1))`.
   - Time point that is neither the first nor last time point — The duration between the midpoint of the previous time interval to the midpoint of the subsequent time interval `((t(k + 1) - t(k))/2 + (t(k) - t(k - 1))/2)`.
   - Last time point — The duration of the last time interval `((t(end) - t(end - 1))`.
2. Normalizing the weighting for each time by dividing each weighting by the mean of all weightings.

   Note If the `timeseries` object is uniformly sampled, then the normalized weighting for each time is 1.0. Therefore, time weighting has no effect.

3. Multiplying the data for each time by its normalized weighting.

See Also

`iqr` | `max` | `mean` | `min` | `std` | `sum` | `timeseries` | `var`

timeseries.min

Purpose

Minimum value of `timeseries` data

Syntax

```
ts_min = min(ts)
ts_min = method(ts,Name,Value)
```

Description

`ts_min = min(ts)` returns the minimum value in the `timeseries` data.

`ts_min = method(ts,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

ts

The `timeseries` object for which you want the minimum data value.

Name-Value Pair Arguments

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

'MissingData'

A string specifying one of two possible values, `remove` or `interpolate`, indicating how to treat missing data during the calculation.

Default: `remove`

'Quality'

A vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).

Output Arguments

ts_min

The minimum value of `ts.Data`, as follows:

- When `ts.Data` is a vector, `ts_min` is the minimum value of `ts.Data` values.
- When `ts.Data` is a matrix, and `IsTimeFirst` is `true`, and the first dimension of `ts` is aligned with time, then `ts_min` is a row vector containing the minimum value of each column of `ts.Data`.

When `ts.Data` is an N-dimensional array, `min` always operates along the first nonsingleton dimension of `ts.Data`.

Examples

The following example finds the minimum values in multivariate time-series data. MATLAB finds the minimum independently for each data column in the `timeseries` object.

```
% Load a 24-by-3 data array:

load count.dat

% Create a timeseries object with 24 time values:

count_ts = timeseries(count,[1:24],'Name','CountPerSecond');

% Find the minimum in each data column for this timeseries object:

min(count_ts)
```

MATLAB returns:

```
7     9     7
```

See Also

`iqr` | `max` | `mean` | `median` | `std` | `sum` | `timeseries` | `var`

timeseries.plot

Purpose

Plot time series

Syntax

```
plot(ts)
plot(tsc.tsname)
plot(ts,linespec)
plot(tsc.tsname,linespec)
plot(ts,Name, Value)
plot(tsc.tsname,Name, Value)
```

Description

`plot(ts)` plots the `timeseries` data `ts` against time and interpolates values between samples by using either zero-order-hold (`'zoh'`) or linear interpolation (the default). The plot displays in the current axes. MATLAB creates a title and axes, if none exists.

`plot(tsc.tsname)` plots the `timeseries` object, `tsname` that is part of the `tscollection`, `tsc`.

`plot(ts,linespec)` plots the `timeseries` data using a line graph and applies the specified `linespec` to lines, markers, or both.

`plot(tsc.tsname,linespec)` plots the `timeseries` object that is part of a `timeseries` collection as a line graph and applies the specified `linespec` to lines, markers, or both.

`plot(ts,Name, Value)` plots a line graph of the time series data using the values specified for `lineseries properties`.

`plot(tsc.tsname,Name, Value)` plots a line graph of the `timeseries` object that is part of the specified `timeseries` collection using the values specified for `lineseries properties`.

Tips

- The `timeseries/plot` method generates titles and axis labels automatically. These labels are:
  - Plot Title — `'Time Series Plot:  <name>'`

    where `<name>` is the string assigned to `ts.Name`, or by default, `'unnamed'`
  - X-Axis Label — `'Time (<units>)'`

where <units> is the value of the ts.TimeInfo.Units field, which defaults to 'seconds'

- Y-Axis Label — '<name>'

  where <name> is the string assigned to ts.Name, or by default, 'unnamed'

- You can place new time series data on a time series plot (by setting hold on, for example, and issuing another timeseries/plot command). When you add data to a plot, the title and axis labels become blank strings to avoid labeling confusion. You can add your own labels after plotting using the title, xlabel, and ylabel commands.
- Time series events, when defined, are marked in the plot with a circular marker with red fill. You can also specify markers for all data points using a linespec or name/value syntax in addition to any event markers your data defines. The event markers plot on top of the markers you define.
- The value assigned to ts.DataInfo.Interpolation.Name controls the type of interpolation the plot method uses when plotting and resampling time series data. Invoke the timeseries method setinterpmethod to change default linear interpolation to zero-order hold interpolation (staircase). This method creates a new timeseries object, with which you can overwrite the original one if you want. For example, to cause time series ts to use zero-order hold interpolation, type the following:

```
ts = ts.setinterpmethod('zoh');
```

Input Arguments

ts

A timeseries object.

tsc

A tscollection.

tsname

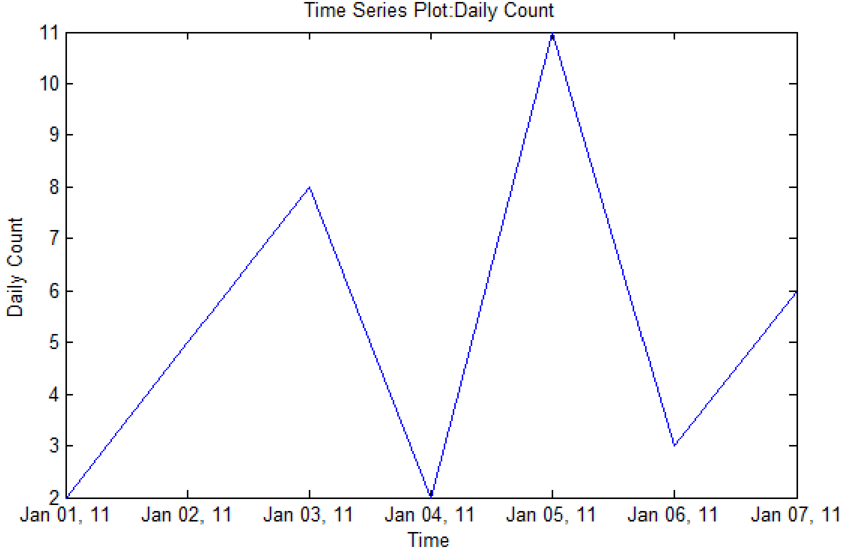
The name of a `timeseries` object within the `tscollection` .

Examples

Plot Time Series Object with Specified Start Date

Create a time series object, set the start date, and then plot the time vector relative to the start date.

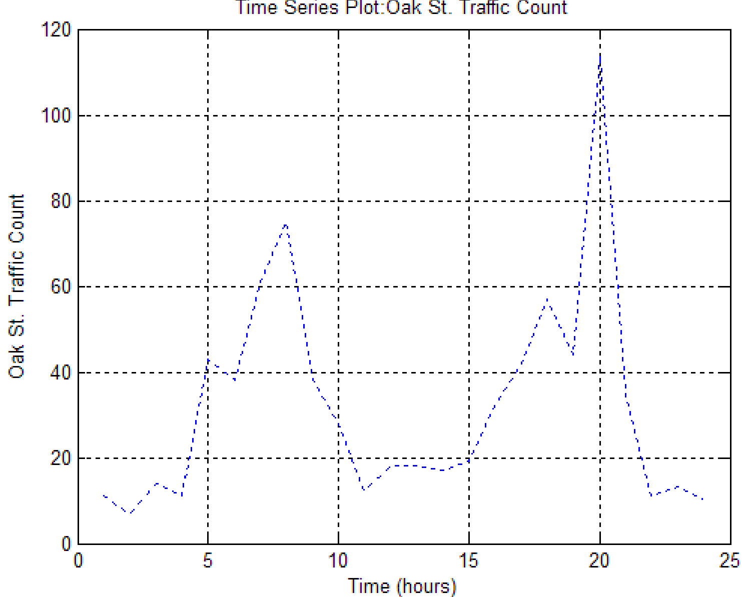
```
x = [2 5 8 2 11 3 6];
ts1 = timeseries(x,1:7);
ts1.Name = 'Daily Count';
ts1.TimeInfo.Units = 'days';
ts1.TimeInfo.StartDate='01-Jan-2011' % Set start date.
ts1.TimeInfo.Format = 'mmm dd, yy'   % Set format for display on x-axis.
ts1.Time=ts1.Time-ts1.Time(1);       % Express time relative to the start date.
plot(ts1)
```



Plot Two Time Series Objects on the Same Axes

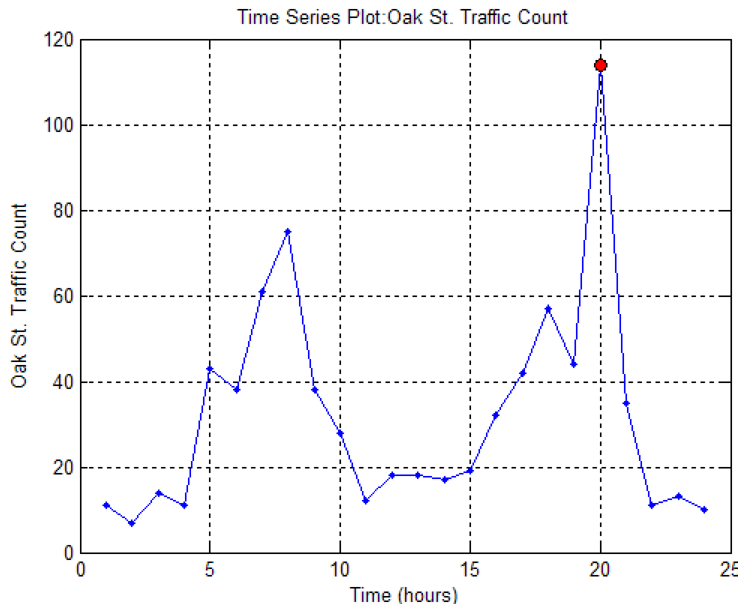
Create two time series objects from traffic count data, and then plot them in sequence on the same axes. Add an event to one series, which is automatically displayed with a red marker.

```
load count.dat;
count1=timeseries(count(:,1),1:24);
count1.Name = 'Oak St. Traffic Count';
count1.TimeInfo.Units = 'hours';
plot(count1,':b'), grid on
```

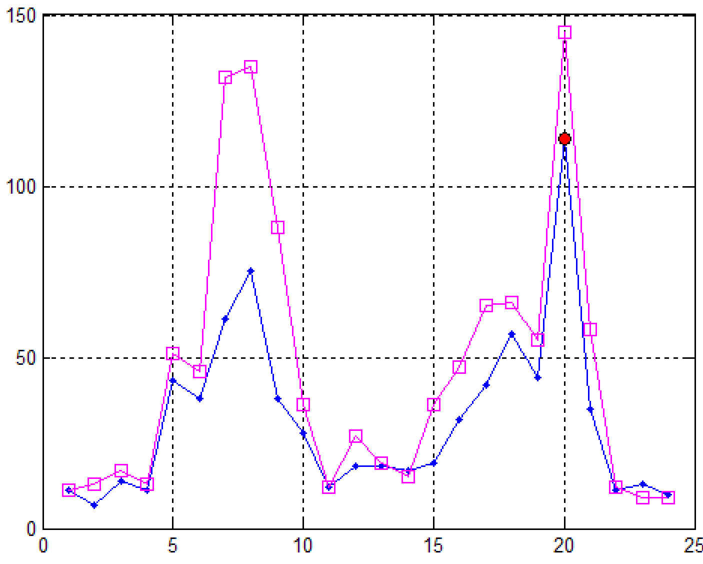



```
% Obtain time of maximum value and add it as an event:
[~,index] = max(count1.Data);
max_event = tsdata.event('peak',count1.Time(index));
max_event.Units = 'hours';
% Add the event to the time series:
count1 = addevent(count1,max_event);
```

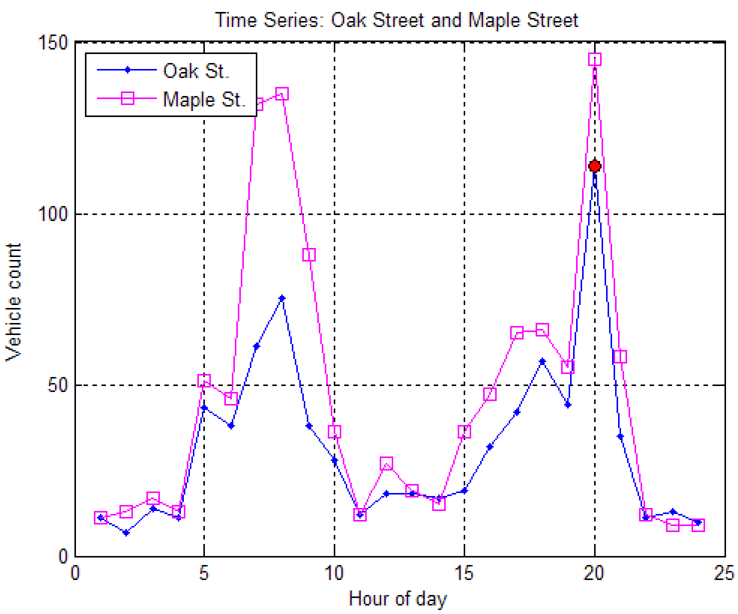
```
% Replace plot with new one showing the event:
plot(count1,'.-b'), grid on
```




```
% Make a new ts object from column 2 of the same data source:
count2=timeseries(count(:,2),1:24);
count2.Name = 'Maple St. Traffic Count';
count2.TimeInfo.Units = 'Hours';
% Turn hold on to add the new data to the plot:
hold on
% The plot method does not add labels to a held plot.
% Use property/value pair to customize markers:
plot(count2,'s-m','MarkerSize',6),
```




```
% Labels are erased, so generate them manually:
title('Time Series: Oak Street and Maple Street')
xlabel('Hour of day')
ylabel('Vehicle count')
% Add a legend in the upper left:
legend('Oak St.','Maple St.','Location','northwest')
```



See Also

`timeseries` | `setinterpmethod` | `tscollection` | `tsdata.event` | `plot`

Purpose

Select or interpolate `timeseries` data using new time vector

Syntax

```
ts1 = resample(ts, time)
ts1 = resample(ts, time, interp_method)
ts1 = resample(ts, time, interp_method, code)
```

Description

`ts1 = resample(ts, time)` resamples the `timeseries` object, `ts`, using the new `time` vector. The `resample` method uses the default interpolation method, which you can view by using the `getinterpmethod(ts)` syntax.

`ts1 = resample(ts, time, interp_method)` resamples the `timeseries` object `ts` using the specified interpolation method, `interp_method`.

`ts1 = resample(ts, time, interp_method, code)` resamples the `timeseries` object `ts` using the interpolation method given by the string `interp_method`. MATLAB applies the `code` to all samples.

Input Arguments

ts

The `timeseries` object that you want to resample.

time

The time vector you want to use to resample the `timeseries` object.

When `ts` uses date strings and `time` is numeric, then `time` is treated as specified relative to the `ts.TimeInfo.StartDate` property and in the same units that `ts` uses.

interp_method

A string specifying the interpolation method. Valid interpolation methods are `linear` and zero-order hold, `zoh` .

Default: `linear`

code

An integer value that specifies the user-defined `Quality` code for resampling. MATLAB applies this `Quality` code to all samples.

Output Arguments

ts1

The `timeseries` object that results when you interpolate the original `timeseries` object with a new time vector.

Examples

This example shows how to resample a `timeseries` object.

Create a `timeseries` object.

```
ts1 = timeseries([1.1; 2.9; 3.7; 4.0; 3.0],1:5,'Name','speed');
```

View the time, data, and interpolation method.

```
ts1.time
ts1.data
ts1.getinterpmethod
```

Resample `ts1` using its default interpolation method.

```
res_ts=resample(ts1,[1 1.5 3.5 4.5 4.9]);
```

View the time, data, and interpolation method for the resampled object.

```
res_ts.time
res_ts.data
res_ts.getinterpmethod
```

See Also

`timeseries` | `getinterpmethod` | `setinterpmethod` | `synchronize`

Purpose

Set properties of `timeseries` object

Syntax

```
set(ts, PropertyName, Value)
set(ts, PropertyName)
set(ts)
```

Description

`set(ts, PropertyName, Value)` sets the named property, `Name`, of the `timeseries` object, `ts`, to the value, `Value`. The following syntax is equivalent:

```
ts.Property = Value
```

`set(ts, PropertyName)` displays the value of the named property for the `timeseries` object, `ts`.

`set(ts)` displays all properties and values of the `timeseries` object `ts`.

Input Arguments

ts

A `timeseries` object.

PropertyName

A string specifying the name of a `timeseries` property. For a list of `timeseries` properties, see `timeseries`.

Value

The value to which you want to set the named property.

Examples

Create a `timeseries`, set its name to `mytimeseries`, and then view the `timeseries` properties:

```
ts1 = timeseries(rand(5,1),[1 2 3 4 5]);
set(ts1, 'Name', 'mytimeseries')
set(ts1)
```

See Also

`timeseries` | `get`

timeseries.setabstime

Purpose

Set times of `timeseries` object as date strings

Syntax

```
ts1=setabstime(ts, times)
ts1=setabstime(ts, times, format)
```

Description

`ts1=setabstime(ts, times)` sets the times in `ts` to the date strings specified in `times`.

`ts1=setabstime(ts, times, format)` explicitly specifies the date-string format, `format`, used in `times`.

Input Arguments

ts

The `timeseries` object for which you want to set times as date strings.

times

A cell array of strings or a `char` array containing valid date or time values in the same date format.

format

The date-string format used for the time values.

Output Arguments

ts1

The `timeseries` object that results from setting times as date strings on the original `timeseries` object.

Examples

Create a `timeseries` object, and then set the absolute time vector:

```
ts = timeseries(rand(3,1))
ts1 = setabstime(ts,{'12-DEC-2005 12:34:56',...
     '12-DEC-2005 13:34:56','12-DEC-2005 14:34:56'})

% View each timeseries object in the Variable Edtior to see the
% differences in the time vectors for each.
```

See Also

`datestr` | `getabstime` | `timeseries`

timeseries.setinterpmethod

Purpose

Set default interpolation method for `timeseries` object

Syntax

```
ts = setinterpmethod(ts, method)
ts = setinterpmethod(ts, fhandle)
ts = setinterpmethod(ts, interpobj)
```

Description

`ts = setinterpmethod(ts, method)` sets the default interpolation method, `method`, for `timeseries` object, `ts`, and outputs it to `ts1`.

`ts = setinterpmethod(ts, fhandle)` sets the default interpolation method for `timeseries` object `ts`, where `fhandle` is a function handle to the interpolation method.

`ts = setinterpmethod(ts, interpobj)` sets the default interpolation method for `timeseries` object `ts`, where `interpobj` is a `tsdata.interpolation` object that directly replaces the interpolation object stored in `ts`.

Tips

- This method is case sensitive.

Input Arguments

ts

The `timeseries` object for which you want to set the default interpolation method.

method

A string specifying the interpolation method. Valid values are `linear` and zero-order hold, `zoh`.

Default: `linear`

fhandle

A function handle to the interpolation method. The order of input arguments defining the function handle must be `new_time`, `time`, and `data`. The single output argument must be the interpolated data only.

interpobj

timeseries.setinterpmethod

A tsdata.interpolation object a tsdata.interpolation object that directly replaces the interpolation object stored in ts.

Output Arguments

ts1

The timeseries object that results when you set the interpolation method for the original timeseries object.

Examples

Set the default interpolation method for timeseries object ts to zero order hold:

```
ts = timeseries(rand(100,1),1:100);
ts = setinterpmethod(ts,'zoh');
plot(ts);
```

Set the default interpolation method for timeseries object ts, where fhandle is a function handle to the interpolation method defined by function handle myFuncHandle:

```
ts = timeseries(rand(100,1),1:100);
myFuncHandle = @(new_time, time, data)...
               interp1(time, data, new_time,...
                       'linear','extrap');
ts = setinterpmethod(ts, myFuncHandle);
ts = resample(ts, [-5:0.1:10]);
plot(ts);
```

Set the default interpolation method for timeseries object ts to a tsdata.interpolation object:

```
ts = timeseries(rand(100,1),1:100);
myFuncHandle = @(new_time, time, data)...
               interp1(time, data, new_time,...
                       'linear','extrap');
myInterpObj = tsdata.interpolation(myFuncHandle);
ts = setinterpmethod(ts,myInterpObj);
plot(ts);
```

See Also

`timeseries` | `getinterpmethod`

timeseries.setuniformtime

Purpose

Modify uniform time vector of `timeseries` object

Syntax

```
ts2 = setuniformtime(ts1,'StartTime',StartTime)
ts2 = setuniformtime(ts1,'Interval',Interval)
ts2 = setuniformtime(ts1,'EndTime',EndTime)
ts2 = setuniformtime(ts1,'StartTime',StartTime,'Interval',
   Interval)
ts2 = setuniformtime(ts1,'StartTime',StartTime,'EndTime',
   EndTime)
ts2 = setuniformtime(ts1,'Interval',Interval,'EndTime',
   EndTime)
```

Description

`ts2 = setuniformtime(ts1,'StartTime',StartTime)` returns the time series with a modified uniform time vector, determined from the `StartTime` and `Interval 1. EndTime = StartTime + (length(ts1) - 1)`. The unit of time is unchanged.

`ts2 = setuniformtime(ts1,'Interval',Interval)` sets the `StartTime` to `0`, and uses `EndTime = (length(ts1) - 1)*Interval`.

`ts2 = setuniformtime(ts1,'EndTime',EndTime)` sets the `StartTime` to `0`, and uses `Interval = EndTime/(length(ts1) -1)`.

`ts2 = setuniformtime(ts1,'StartTime',StartTime,'Interval',Interval)` uses `EndTime = StartTime + (length(ts1) - 1) * Interval`.

`ts2 = setuniformtime(ts1,'StartTime',StartTime,'EndTime',EndTime)` uses `Interval = (EndTime - StartTime)/(length(ts1) - 1)`.

`ts2 = setuniformtime(ts1,'Interval',Interval,'EndTime',EndTime)` uses `StartTime = EndTime - (length(ts1) - 1) * Interval`.

Input Arguments

ts1

`timeseries` object to which you want to assign a uniform time vector.

StartTime

Start time of uniform time vector, specified as a numeric value.

Interval

Time interval of uniform time vector, specified as a numeric scalar value.

EndTime

End time of uniform time vector specified as a numeric scalar value.

Output Arguments

ts2

Time series with uniform time vector, returned as a `timeseries` object.

Examples

Specify New Start Time

Modify the uniform time vector of time series data by specifying a new start time.

1 Load the sample data.

```
load count.dat;
```

2 Create a `timeseries` object.

```
count_ts = timeseries(count,1:length(count),'Name','CountPerSecond'
```

3 Modify uniform time vector of `count_ts`.

```
count_ts = setuniformtime(count_ts,'StartTime',10);
```

The start time of the time vector is `10` seconds. `setuniformtime` uses a default time interval of `1` and computes the end time using: `EndTime = StartTime + (length(count_ts) - 1)* Interval`.

Specify New Start and End Times

Modify the uniform time vector of time series data by specifying a new start time and end time.

1 Load sample data.

```
load count.dat;
```

2 Create `timeseries` object.

```
count_ts = timeseries(count,1:length(count),'Name','CountPerSecond');
```

3 Assign a uniform time vector to `count_ts`.

```
count_ts2 = setuniformtime(count_ts,'StartTime',10,'EndTime',20);
```

The start time of the time vector is now 10 seconds, and the end time is now 20 seconds. MATLAB computes the time interval using: `Interval = (EndTime - StartTime)/(length(count_ts) - 1)`

See Also

`timeseries`

Purpose

Synchronize and resample two `timeseries` objects using common time vector

Syntax

```
[ts1 ts2] = synchronize(ts1,ts2, synchronizemethod)
[ts1 ts2] = synchronize(ts1,ts2, Name,Value)
```

Description

`[ts1 ts2] = synchronize(ts1,ts2, synchronizemethod)` creates two new `timeseries` objects by synchronizing `ts1` and `ts2` using a common time vector and the specified method.

`[ts1 ts2] = synchronize(ts1,ts2, Name,Value)` creates the two new `timeseries` objects with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

ts1

One of the `timeseries` objects that you want to synchronize and resample.

ts2

The other `timeseries` object that you want to synchronize and resample.

synchronizemethod

A string that defines the method for synchronizing the `timeseries` object. It can be any one of the following:

- `Union` — Resample `timeseries` objects using a time vector that is a union of the time vectors of `ts1` and `ts2` on the time range where the two time vectors overlap.
- `Intersection` — Resample `timeseries` objects on a time vector that is the intersection of the time vectors of `ts1` and `ts2`.
- `Uniform` — Requires an additional argument as follows:

  ```
  [ts1 ts2] = synchronize(ts1,ts2,'Uniform','Interval',value)
  ```

This method resamples time series on a uniform time vector, where `value` specifies the time interval between two consecutive samples. The uniform time vector is the overlap of the time vectors of `ts1` and `ts2`. The interval units are the smaller units of `ts1` and `ts2`.

Name-Value Pair Arguments

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

'InterpMethod'

Forces the specified interpolation method (over the default method) for this synchronize operation. Can be either a string, `linear` or `zoh`, or a `tsdata.interpolation` object that contains a user-defined interpolation method.

Default: `linear`

'QualityCode'

Integer (between -128 and 127) used as the quality code for both time series after the synchronization.

'KeepOriginalTimes'

Logical value (`true` or `false`) indicating whether the new time series should keep the original time values.

'tolerance'

Real number used as the tolerance for differentiating two time values when comparing the `ts1` and `ts2` time vectors. The default tolerance is `1e-10`. For example, when the sixth time value in `ts1` is `5+(1e-12)` and the sixth time value in `ts2` is `5-(1e-13)`, both values are treated as 5 by default. To differentiate those

two times, you can set 'tolerance' to a smaller value such as 1e-15, for example.

Output Arguments

ts1

One of the timeseries objects that you synchronized and resampled.

ts2

The other timeseries object that you synchronized and resampled.

Examples

This example illustrates how the KeepOriginalTime property affects synchronization.

```
% Create two timeseries, such that ts1.timeinfo.StartDate
% is one day after ts2.timeinfo.StartDate:

ts1 = timeseries([1 2],[datestr(now); datestr(now+1)]);
ts2 = timeseries([1 2],[datestr(now-1); datestr(now)]);

% If you use this code, then ts1.timeinfo.StartDate
% is changed to match ts2.TimeInfo.StartDate
% and ts1.Time changes to 1:

[ts1 ts2] = synchronize(ts1,ts2,'union');

% But if you use this code, then ts1.timeinfo.StartDate
% is unchanged and ts1.Time is still 0:

[ts1 ts2] = synchronize(ts1,ts2,'union','KeepOriginalTimes',true);
```

See Also

set | timeseries

timeseries.transpose

Purpose

Transpose `timeseries` object

Syntax

```
ts1 = transpose(ts)
```

Description

`ts1 = transpose(ts)` returns a new `timeseries` object, `ts1`, with `IsTimeFirst` value set to the opposite of what it is for `ts`. For example, if `ts` has the first data dimension aligned with the time vector, `ts1` has the last data dimension aligned with the time vector.

Tips

- The `transpose` function that is overloaded for `timeseries` objects does not transpose the data. Instead, this function changes whether the first or the last dimension of the data aligns with the time vector. To transpose the data, transpose the `Data` property of the time series. For example, you can use the syntax `transpose(ts.Data)` or `(ts.Data).'`. The value of the `Data` property must be a 2-D array.
- Consider a time series with 10 samples with the property `IsTimeFirst = True`. When you transpose this time series, the data size changes from 10-by-1 to 1-by-1-by-10. Note that the first dimension of the `Data` property is shown explicitly.

  The following table summarizes how the size for timeseries data (up to three dimensions) display before and after transposing.

  Data Size Before and After Transposing

Size of Original Data	Size of Transposed Data
N-by-1	1-by-1-by-N
N-by-M	M-by-1-by-N
N-by-M-by-L	M-by-L-by-N

Input Arguments

ts

The `timeseries` object that you want to transpose.

Output Arguments

ts1

The `timeseries` object that is the result of transposing the original `timeseries` object.

Examples

Suppose that a `timeseries` object, `ts`, has `ts.Data` size 10-by-3-by-2 and its time vector has a length of 10. The `IsTimeFirst` property of `ts` is `true`, which means that the first dimension of the data aligns with the time vector. `transpose(ts)` modifies the `timeseries` object, such that the last dimension of the data now aligns with the time vector. This permutes the data, such that the size of `ts.Data` becomes 3-by-2-by-10.

See Also

`timeseries` | `transpose`

timeseries.std

Purpose

Standard deviation of `timeseries` data

Syntax

```
ts_std = std(ts)
ts_std = std(ts,Name,Value)
```

Description

`ts_std = std(ts)` returns the standard deviation of the `timeseries` data.

`ts_std = std(ts,Name,Value)` specifies additional options specified with one or more `Name,Value` pair arguments.

Input Arguments

ts

The `timeseries` object for which you want the standard deviation of the data.

Name-Value Pair Arguments

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

'MissingData'

A string specifying one of two possible values, `remove` or `interpolate`, indicating how to treat missing data during the calculation.

Default: `remove`

'Quality'

A vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).

'Weighting'

A string specifying one of two possible values, `none` or `time`. When you specify `time`, larger time values correspond to larger weights.

Output Arguments

ts_std

The standard deviation of `ts.Data` values as follows:

- When `ts.Data` is a vector, `ts_std` is the standard deviation of `ts.Data` values.
- When `ts.Data` is a matrix, and `IsTimeFirst` is `true`, and the first dimension of `ts` is aligned with time, then `ts_std` is the standard deviation of each column of `ts.Data`.

When `ts.Data` is an N-dimensional array, `std` always operates along the first nonsingleton dimension of `ts.Data`.

Examples

The following example finds the standard deviation for a `timeseries` object. MATLAB calculates the standard deviation for each data column in the `timeseries` object.

```
% Load a 24-by-3 data array:

load count.dat

% Create a timeseries object with 24 time values:

count_ts = timeseries(count,1:24,'Name','CountPerSecond');

% Calculate the standard deviation of each data column for this
% timeseries object:

std(count_ts)
```

MATLAB returns:

```
25.3703   41.4057   68.0281
```

Algorithms

MATLAB determines weighting by:

1. Attaching a weighting to each time value, depending on its order, as follows:
   - First time point — The duration of the first time interval `(t(2) - t(1))`.
   - Time point that is neither the first nor last time point — The duration between the midpoint of the previous time interval to the midpoint of the subsequent time interval `((t(k + 1) - t(k))/2 + (t(k) - t(k - 1))/2)`.
   - Last time point — The duration of the last time interval `((t(end) - t(end - 1))`.
2. Normalizing the weighting for each time by dividing each weighting by the mean of all weightings.

   Note If the `timeseries` object is uniformly sampled, then the normalized weighting for each time is 1.0. Therefore, time weighting has no effect.

   ---

3. Multiplying the data for each time by its normalized weighting.

See Also

`iqr` | `max` | `mean` | `median` | `min` | `sum` | `timeseries` | `var`

Purpose

Sum of `timeseries` data

Syntax

```
ts_sm = sum(ts)
ts_sm = sum(ts,Name,Value)
```

Description

`ts_sm = sum(ts)` returns the sum of the `timeseries` data.

`ts_sm = sum(ts,Name,Value)` specifies additional options with one or more `Name,Value` pair arguments.

Input Arguments

ts

The `timeseries` object for which you want the sum of the data.

Name-Value Pair Arguments

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

'MissingData'

A string specifying one of two possible values, `remove` or `interpolate`, indicating how to treat missing data during the calculation.

Default: `remove`

'Quality'

A vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).

'Weighting'

A string specifying one of two possible values, `none` or `time`. When you specify `time`, larger time values correspond to larger weights.

Output Arguments

ts_sm

The sum of the `timeseries` data, as follows:

- When `ts.Data` is a vector, ts_sm is the sum of `ts.Data` values.
- When `ts.Data` is a matrix, and `IsTimeFirst` is `true`, and the first dimension of `ts` is aligned with time, then ts_sm is a row vector containing the sum of each column of `ts.Data`.

When `ts.Data` is a N-dimensional array, `sum` always operates along the first nonsingleton dimension of `ts.Data`.

Examples

Calculate the sum of each data column for a `timeseries` object:

```
% Load a 24-by-3 data array:

load count.dat

% Create a timeseries object with 24 time values:

count_ts = timeseries(count,1:24,'Name','CountPerSecond');

% Calculate the sum of each data column for this timeseries object:

sum(count_ts)
```

MATLAB returns:

```
768        1117        1574
```

Algorithms

MATLAB determines weighting by:

1. Attaching a weighting to each time value, depending on its order, as follows:
   - First time point — The duration of the first time interval `(t(2) - t(1))`.
   - Time point that is neither the first nor last time point — The duration between the midpoint of the previous time interval to the midpoint of the subsequent time interval `((t(k + 1) - t(k))/2 + (t(k) - t(k - 1))/2)`.
   - Last time point — The duration of the last time interval `((t(end) - t(end - 1))`.
2. Normalizing the weighting for each time by dividing each weighting by the mean of all weightings.

   Note If the `timeseries` object is uniformly sampled, then the normalized weighting for each time is 1.0. Therefore, time weighting has no effect.

3. Multiplying the data for each time by its normalized weighting.

See Also

`iqr` | `max` | `mean` | `median` | `min` | `std` | `timeseries` | `var`

timeseries.var

Purpose

Variance of `timeseries` data

Syntax

```
ts_var = var(ts)
ts_var = var(ts,Name,Value)
```

Description

`ts_var = var(ts)` returns the variance of `ts.data`.

`ts_var = var(ts,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

ts

The `timeseries` object for which you want the variance of the data.

Name-Value Pair Arguments

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

'MissingData'

A string specifying one of two possible values, `remove` or `interpolate`, indicating how to treat missing data during the calculation.

Default: `remove`

'Quality'

A vector of integers, indicating which quality codes represent missing samples (for vector data) or missing observations (for data arrays with two or more dimensions).

'Weighting'

A string specifying one of two possible values, none or time. When you specify time, larger time values correspond to larger weights.

Output Arguments

ts_var

The variance of ts.data, as follows:

- When ts.Data is a vector, then ts_var is the variance of ts.Data values.
- When ts.Data is a matrix, and IsTimeFirst is true, and the first dimension of ts is aligned with time, then ts_var is a row vector containing the variance of each column of ts.Data.

When ts.Data is an N-dimensional array, var always operates along the first nonsingleton dimension of ts.Data.

Examples

The following example calculates the variance values of a multivariate timeseries object. MATLAB calculates the variance independently for each data column in the timeseries object.

Load a 24-by-3 data array. Then create a timeseries object with 24 time values.

```
load count.dat
count_ts = timeseries(count,[1:24],'Name','CountPerSecond');
```

Calculate the variance of each data column.

```
var(count_ts)
```

MATLAB returns:

```
1.0e+03 *
  0.6437    1.7144    4.6278
```

Algorithms

MATLAB determines weighting by:

1. Attaching a weighting to each time value, depending on its order, as follows:
   - First time point — The duration of the first time interval `(t(2) - t(1))`.
   - Time point that is neither the first nor last time point — The duration between the midpoint of the previous time interval to the midpoint of the subsequent time interval `((t(k + 1) - t(k))/2 + (t(k) - t(k - 1))/2)`.
   - Last time point — The duration of the last time interval `((t(end) - t(end - 1))`.
2. Normalizing the weighting for each time by dividing each weighting by the mean of all weightings.

   Note If the `timeseries` object is uniformly sampled, then the normalized weighting for each time is 1.0. Therefore, time weighting has no effect.

   ---

3. Multiplying the data for each time by its normalized weighting.

See Also

`iqr` | `max` | `mean` | `median` | `min` | `std` | `timeseries` | `sum`

Purpose

Triangulation in 2-D or 3-D

Description

Use `triangulation` to create an in-memory representation of any 2-D or 3-D triangulation data that is in matrix format, such as the matrix output from the `delaunay` function or other software tools. When your data is represented using `triangulation`, you can perform topological and geometric queries, which you can use to develop geometric algorithms. For example, you can find the triangles or tetrahedra attached to a vertex, those that share an edge, their circumcenters, and other features.

Construction

`TR = triangulation(T,P)` creates a 2-D or 3-D triangulation representation using the triangulation connectivity list, `T`, and the points in matrix `P`.

`TR = triangulation(T,x,y)` creates a 2-D triangulation representation with the point coordinates specified as column vectors, `x` and `y`.

`TR = triangulation(T,x,y,z)` creates a 3-D triangulation representation with the point coordinates specified as column vectors, `x`, `y`, and `z`.

Input Arguments

T

Triangulation connectivity list, specified as an `m`-by-`n` matrix, where `m` is the number of triangles or tetrahedra, and `n` is the number of vertices per triangle or tetrahedron. Each element in `T` is a “Vertex ID” on page 1-4317. Each row of `T` contains the vertex IDs that define a triangle or tetrahedron.

P

Points, specified as a matrix whose columns are the `x`, `y`, (and possibly `z`) coordinates of the triangulation points. The row numbers of `P` are the vertex IDs in the triangulation.

x

x-coordinates vector, specified as a column vector containing the *x*-coordinates of the triangulation points.

y

y-coordinates vector, specified as a column vector containing the *y*-coordinates of the triangulation points.

z

z-coordinates vector, specified as a column vector containing the *z*-coordinates of the triangulation points.

Properties

Points

Points in the triangulation, represented as a matrix containing the following information:

- Each row in `TR.Points` contains the coordinates of a vertex.
- Each row number of `TR.Points` is a vertex ID.

ConnectivityList

Triangulation connectivity list, represented as a matrix. This matrix contains the following information:

- Each element in `TR.ConnectivityList` is a vertex ID.
- Each row represents a triangle or tetrahedron in the triangulation.
- Each row number of `TR.ConnectivityList` is a "Triangle or Tetrahedron ID" on page 1-4317.

Methods

barycentricToCartesian	Converts point coordinates from barycentric to Cartesian
cartesianToBarycentric	Converts point coordinates from Cartesian to barycentric

circumcenter	Circumcenter of triangle or tetrahedron
edgeAttachments	Triangles or tetrahedra attached to specified edge
edges	Triangulation edges
faceNormal	Triangulation face normal
featureEdges	Triangulation sharp edges
freeBoundary	Triangulation facets referenced by only one triangle or tetrahedron
incenter	Incenter of triangle or tetrahedron
isConnected	Test if two vertices are connected by edge
neighbors	Neighbors to specified triangle or tetrahedron
size	Size of triangulation connectivity list
vertexAttachments	Triangles or tetrahedra attached to specified vertex
vertexNormal	Triangulation vertex normal

Definitions

Vertex ID

A row number of the matrix, `TR.Points`. Use this ID to refer a specific vertex in the triangulation.

Triangle or Tetrahedron ID

A row number of the matrix, `TR.ConnectivityList`. Use this ID to refer a specific triangle or tetrahedron.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

2-D Triangulation

Define the points in the triangulation.

```
P = [ 2.5    8.0
      6.5    8.0
      2.5    5.0
      6.5    5.0
      1.0    6.5
      8.0    6.5];
```

Define the triangles. This is the triangulation connectivity list.

```
T = [5  3  1;
     3  2  1;
     3  4  2;
     4  6  2];
```

Create the triangulation representation.

```
TR = triangulation(T,P)
```

```
TR =

  triangulation with properties:

              Points: [6x2 double]
    ConnectivityList: [4x3 double]
```

Examine the coordinates of the vertices of the first triangle.

```
TR.Points(TR.ConnectivityList(1,:),:)
```

```
ans =

    1.0000    6.5000
```

```
    2.5000    5.0000
    2.5000    8.0000
```

See Also

`delaunayTriangulation` |

triangulation.barycentricToCartesian

Purpose

Converts point coordinates from barycentric to Cartesian

Syntax

```
PC = barycentricToCartesian(TR,ti,B)
```

Description

`PC = barycentricToCartesian(TR,ti,B)` returns the Cartesian coordinates of the points in `B`. Each row, `B(j,:)`, contains the barycentric coordinates of a point with respect to the triangle or tetrahedron, `ti(j)`. The point, `PC(j,:)`, is the `j`th point represented in Cartesian coordinates.

Input Arguments

TR

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

ti

Triangle or tetrahedron IDs, specified as a column vector.

B

Barycentric coordinates, specified as a matrix. Each row, `B(j,:)`, contains the barycentric coordinates of a point with respect to the triangle or tetrahedron, `ti(j)`.

Output Arguments

PC

Cartesian coordinates, returned as a matrix. The point, `PC(j,:)`, is the `j`th point represented in Cartesian coordinates.

Definitions

Triangle or Tetrahedron ID

A row number of the matrix, `TR.ConnectivityList`. Use this ID to refer a specific triangle or tetrahedron.

Examples

Barycentric Coordinates Converted to Cartesian Coordinates

Create a triangulation from a set of points, `P`, and triangulation connectivity list, `T`.

```
P = [ 2.5    8.0
      6.5    8.0
      2.5    5.0
      6.5    5.0
      1.0    6.5
      8.0    6.5];

T = [5  3  1;
     3  2  1;
     3  4  2;
     4  6  2];

TR = triangulation(T,P);
```

Specify the first triangle.

```
ti = 1;
```

Specify the barycentric coordinates of the second point in the triangle.

```
B = [0 1 0];
```

Convert the point to Cartesian coordinates.

```
PC = barycentricToCartesian(TR,ti,B)
```

```
PC =

    2.5000    5.0000
```

Mapped Incenters of Deformed Triangulation

Create a Delaunay triangulation from a set of points.

```
x = [0 4 8 12 0 4 8 12]';
y = [0 0 0 0 8 8 8 8]';
DT = delaunayTriangulation(x,y);
```

Calculate the Cartesian coordinates of the incenters.

```
cc = incenter(DT);
```

Plot the original triangulation and reference points.

```
figure
subplot(1,2,1);
triplot(DT);
hold on;
plot(cc(:,1),cc(:,2),'*r');
hold off;
axis equal;
```

Create a new triangulation which is a deformed version of `DT` .

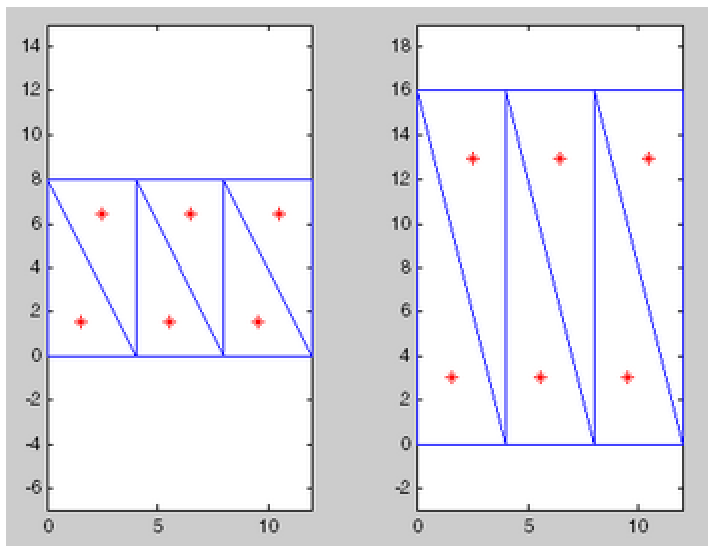
```
ti = DT.ConnectivityList;
y = [0 0 0 0 16 16 16 16]';
TR = triangulation(ti,x,y);
```

Compute the barycentric coordinates for the incenters of `DT`, and use them to compute the Cartesian coordinates of the analogous points in `TR`.

```
b = cartesianToBarycentric(DT,[1:length(ti)]',cc);
xc = barycentricToCartesian(TR,[1:length(ti)]',b);
```

Plot the deformed triangulation and mapped locations of the reference points.

```
subplot(1,2,2);
triplot(TR);
hold on;
plot(xc(:,1),xc(:,2),'*r');
hold off;
axis equal;
```



See Also

`cartesianToBarycentric` | `delaunayTriangulation` |

triangulation.cartesianToBarycentric

Purpose

Converts point coordinates from Cartesian to barycentric

Syntax

```
B = cartesianToBarycentric(TR,ti,PC)
```

Description

`B = cartesianToBarycentric(TR,ti,PC)` returns the barycentric coordinates of the points in `PC`. Each row, `PC(j,:)`, contains the Cartesian coordinates of a point you want to convert. `B(j,:)` are the barycentric coordinates of the point, `PC(j,:)`, with respect to triangle or tetrahedron, `ti(j)`.

Input Arguments

TR

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

ti

Triangle or tetrahedron IDs, specified as a column vector.

PC

Cartesian coordinates, specified as a matrix. Each row, `PC(j,:)`, contains the Cartesian coordinates of a point with respect to triangle or tetrahedron, `ti(j)`.

Output Arguments

B

Barycentric coordinates, returned as a matrix. `B(j,:)` are the barycentric coordinates of `PC(j,:)` with respect to `ti(j)`.

Definitions

Triangle or Tetrahedron ID

A row number of the matrix, `TR.ConnectivityList`. Use this ID to refer a specific triangle or tetrahedron.

Examples

Cartesian Coordinates Coverted to Barycentric Coordinates

Create a triangulation from a set of points, `P`, and triangulation connectivity list, `T`.

```
P = [ 2.5    8.0
      6.5    8.0
      2.5    5.0
      6.5    5.0
      1.0    6.5
      8.0    6.5];

T = [5  3  1;
     3  2  1;
     3  4  2;
     4  6  2];

TR = triangulation(T,P);
```

Specify the first triangle.

```
ti = 1;
```

Get the coordinates of the third vertex in the first triangle.

```
PC = TR.Points(TR.ConnectivityList(1,3),:)

PC =

    2.5000    8.0000
```

Convert the point to barycentric coordinates.

```
B = cartesianToBarycentric(TR,ti,PC)

B =

     0     0     1
```

Mapped Incenters of Deformed Triangulation

Create a Delaunay triangulation from a set of points.

```
x = [0 4 8 12 0 4 8 12]';
```

```
y = [0 0 0 0 8 8 8 8]';
DT = delaunayTriangulation(x,y);
```

Calculate the Cartesian coordinates of the incenters.

```
cc = incenter(DT);
```

Plot the original triangulation and reference points.

```
figure
subplot(1,2,1);
triplot(DT);
hold on;
plot(cc(:,1),cc(:,2),'*r');
hold off;
axis equal;
```

Create a new triangulation which is a deformed version of `DT` .

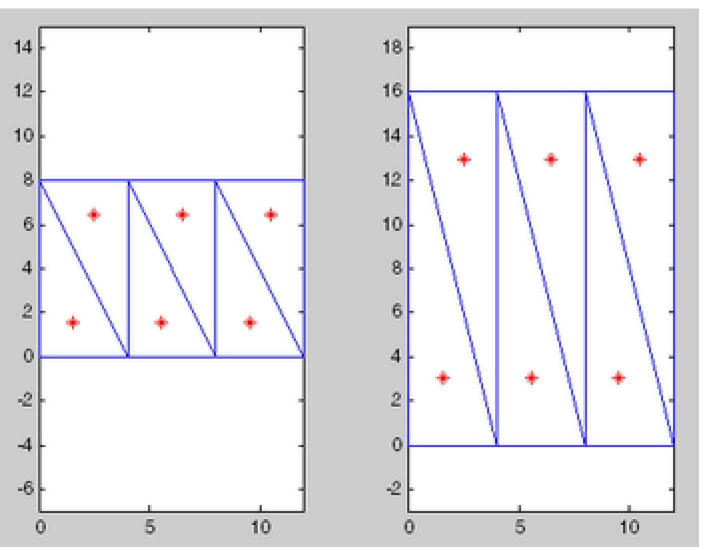
```
ti = DT.ConnectivityList;
y = [0 0 0 0 16 16 16 16]';
TR = triangulation(ti,x,y);
```

Compute the barycentric coordinates for the incenters of `DT`, and use them to compute the Cartesian coordinates of the analogous points in `TR`.

```
b = cartesianToBarycentric(DT,[1:length(ti)]',cc);
xc = barycentricToCartesian(TR,[1:length(ti)]',b);
```

Plot the deformed triangulation and mapped locations of the reference points.

```
subplot(1,2,2);
triplot(TR);
hold on;
plot(xc(:,1),xc(:,2),'*r');
hold off;
axis equal;
```



See Also

`barycentricToCartesian` | `delaunayTriangulation` |

triangulation.circumcenter

Purpose

Circumcenter of triangle or tetrahedron

Syntax

```
CC = circumcenter(TR,ti)
[CC,r] = circumcenter(TR,ti)
CC = circumcenter(TR)
[CC,r] = circumcenter(TR)
```

Description

`CC = circumcenter(TR,ti)` returns the coordinates of the circumcenter of each triangle or tetrahedron specified in `ti`.

`[CC,r] = circumcenter(TR,ti)` also returns the corresponding radii of the circumscribed circles or spheres.

`CC = circumcenter(TR)` returns the circumcenters of all triangles or tetrahedra in the triangulation.

`[CC,r] = circumcenter(TR)` also returns the corresponding radii of the circumscribed circles or spheres for the entire triangulation.

Input Arguments

TR

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

ti

Triangle or tetrahedron IDs, specified as a column vector.

Output Arguments

CC

Circumcenters, returned as a matrix. Each row, `CC(j,:)`, contains the coordinates of the circumcenter of `ti(j)`.

r

Radii of the circumscribed circles or spheres, returned as a vector. The radius at `r(j)` corresponds to the circle or sphere circumscribing `ti(j)`.

Definitions

Triangle or Tetrahedron ID

A row number of the matrix, `TR.ConnectivityList`. Use this ID to refer a specific triangle or tetrahedron.

Examples

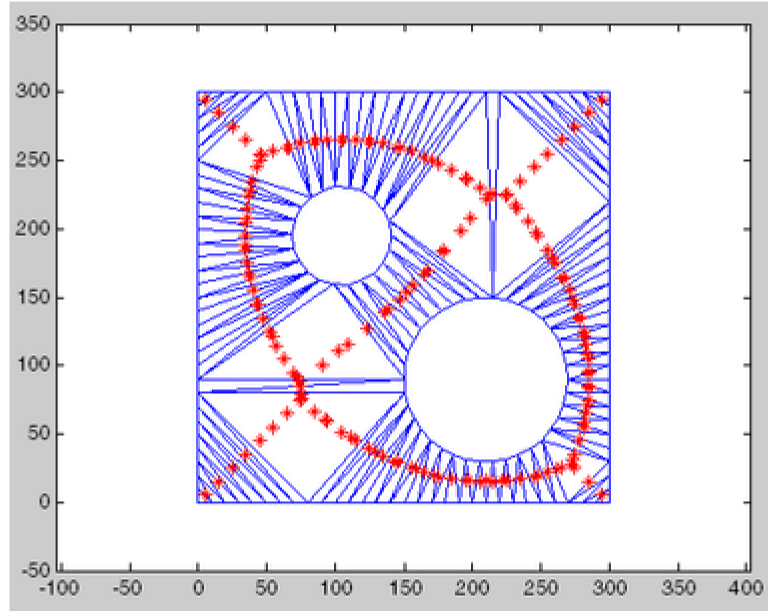
Circumcenters in 2-D Triangulation

Load 2-D triangulation data and create a triangulation representation.

```
load trimesh2d
TR = triangulation(tri,x,y);
```

Compute the circumcenters.

```
CC = circumcenter(TR);
triplot(TR)
axis([-50 350 -50 350]);
axis equal;
hold on;
plot(CC(:,1),CC(:,2),'*r');
hold off;
```



The circumcenters represent points on the medial axis of the polygon.

Circumcenters in 3-D Delaunay Triangulation

Create the Delaunay triangulation using a random set of points, `P`.

```
P = gallery('uniformdata',10,3,0);
DT = delaunayTriangulation(P);
```

Calculate the circumcenters of the first five tetrahedra in `DT`.

```
CC = circumcenter(DT,[1:5]')
```

```
CC =

    0.9626    0.3892    0.0928
    6.3458    0.2377    3.1814
    0.4820    0.9064    0.5176
```

```
   -1.2993    1.8384   -1.2185
   -0.1595    1.0852   -0.2536
```

See Also

`incenter` | `delaunayTriangulation` |

triangulation.edgeAttachments

Purpose

Triangles or tetrahedra attached to specified edge

Syntax

```
ti = edgeAttachments(TR,vstart,vend)
ti = edgeAttachments(TR,E)
```

Description

`ti = edgeAttachments(TR,vstart,vend)` returns the triangles or tetrahedra attached to the specified edges. To specify the edges, use the vectors, `vstart` and `vend`. These vectors contain the "Vertex ID" on page 1-4333 at the start and end of each edge.

`ti = edgeAttachments(TR,E)` specifies the starting and ending vertices of each edge in a 2-column matrix, `E`.

Input Arguments

TR

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

vstart

IDs of starting vertices, specified as a column vector of vertex IDs. Each ID refers to a vertex at the start of an edge.

vend

IDs of ending vertices, specified as a column vector of vertex IDs. Each ID refers to a vertex at the end of an edge.

E

IDs of the edge vertices, specified as a 2-column matrix of vertex IDs. Each row of `E` corresponds to an edge and contains two IDs:

- `E(j,1)` is the ID of the vertex at the start of an edge.
- `E(j,2)` is the ID of the vertex at end of the edge.

Output Arguments

ti

IDs of the triangles or tetrahedra attached to the edges, returned as an `m`-by-`1` cell array. Each cell in `ti` contains the IDs of the

attached triangles or tetrahedra. The attachments to the edge located between the vertices, `vstart(j)` and `vend(j)`, or `E(j,:)`, are returned in `ti{j}`. The attachments are returned in a cell array because the number of triangles or tetrahedra associated with each edge can vary.

Definitions

Vertex ID

A row number of the matrix, `TR.Points`. Use this ID to refer a specific vertex in the triangulation.

Triangle or Tetrahedron ID

A row number of the matrix, `TR.ConnectivityList`. Use this ID to refer a specific triangle or tetrahedron.

Examples

Edge Attachments in 3-D Triangulation

Load 2-D triangulation data and create a triangulation representation.

```
load tetmesh
TR = triangulation(tet,X);
```

Define the starting and ending vertices of the edges.

```
vstart = [15; 21];
vend = [936; 716];
```

Find the edge attachments.

```
ti = edgeAttachments(TR,vstart,vend);
```

Examine the attachments to the first edge.

```
ti{1}
```

```
ans =

     927     2060     3438     3423     2583     4690
```

Edge Attachments in 2-D Delaunay Triangulation

Create a Delaunay triangulation.

```
x = [0 1 1 0 0.5]';
y = [0 0 1 1 0.5]';
DT = delaunayTriangulation(x,y);
```

Find the triangles attached to edge (1,5).

```
ti = edgeAttachments(DT,1,5);
ti{:}
```

```
ans =

     4     1
```

See Also

edges | vertexAttachments | delaunayTriangulation |

triangulation.edges

Purpose

Triangulation edges

Syntax

```
E = edges(TR)
```

Description

`E = edges(TR)` returns the triangulation edges as a matrix of vertex IDs. The vertices referenced in `E` indicate the beginning and end of each edge.

Input Arguments

TR

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

Output Arguments

E

Edge vertex IDs, returned as a 2-column matrix. Each row of `E` corresponds to an edge and contains two IDs:

- `E(j,1)` is the ID of the vertex at the start of the edge.
- `E(j,2)` is the ID of the vertex at end of the edge.

Definitions

Vertex ID

A row number of the matrix, `TR.Points`. Use this ID to refer a specific vertex in the triangulation.

Examples

Edges in a 2-D Triangulation

Load 2-D triangulation data and create a triangulation representation.

```
load trimesh2d
TR = triangulation(tri,x,y);
```

Find the edges in the triangulation.

```
E = edges(TR);
```

List the first five edges.

```
E(1:5,:)

ans =

     1     2
     1   118
     1   119
     1   120
     2     3
```

Edges in a 2-D Delaunay Triangulation

Create 2-D Delaunay triangulation from a set of 10 random points.

```
X = gallery('uniformdata',[10, 2],0);
DT = delaunayTriangulation(X);
```

Find the edges in the triangulation.

```
E = edges(DT)

E =

     1     3
     1     4
     1     5
     1     6
     1     9
     2     4
     2     5
     2     6
     2     7
     2     8
     2    10
     3     4
     3     7
     3    10
     4     6
     4    10
```

```
     5     6
     5     8
     5     9
     6     9
     7     8
     7    10
```

See Also

`edgeAttachments` | `delaunayTriangulation` |

triangulation.faceNormal

Purpose

Triangulation face normal

Syntax

```
FN = faceNormal(TR,ti)
FN = faceNormal(TR)
```

Description

`FN = faceNormal(TR,ti)` returns the unit normal vector to each triangle specified by `ti`. The `faceNormal` function supports only 2-D triangulations.

`FN = faceNormal(TR)` returns the unit normal vectors to all triangles.

Input Arguments

TR

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

ti

Triangle IDs, specified as a column vector.

Output Arguments

FN

Face normals, returned as a matrix. Each row, `FN(j,:)`, is the unit normal vector to triangle `ti(j)`.

If you do not specify `ti`, then `faceNormal` returns the unit normal information for the entire triangulation. In this case, the normal associated with `TR.ConnectivityList(j,:)` is `FN(j,:)`.

Definitions

Triangle ID

A row number of the matrix, `TR.ConnectivityList`. You use this ID to refer a specific triangle.

Examples

Plot Unit Normals to Facets on a Spherical Surface

Create a set of random points on a spherical surface.

```
theta = gallery('uniformdata',[100,1],0)*2*pi;
phi = gallery('uniformdata',[100,1],1)*pi;
```

```
x = cos(theta).*sin(phi);
y = sin(theta).*sin(phi);
z = cos(phi);
```

Triangulate the points with `delaunayTriangulation`.

```
DT = delaunayTriangulation(x,y,z);
```

Find the free boundary facets and use them to create a triangulation representation for plotting.

```
[T,Xb] = freeBoundary(DT);
TR = triangulation(T,Xb);
```

Plot the triangulation.

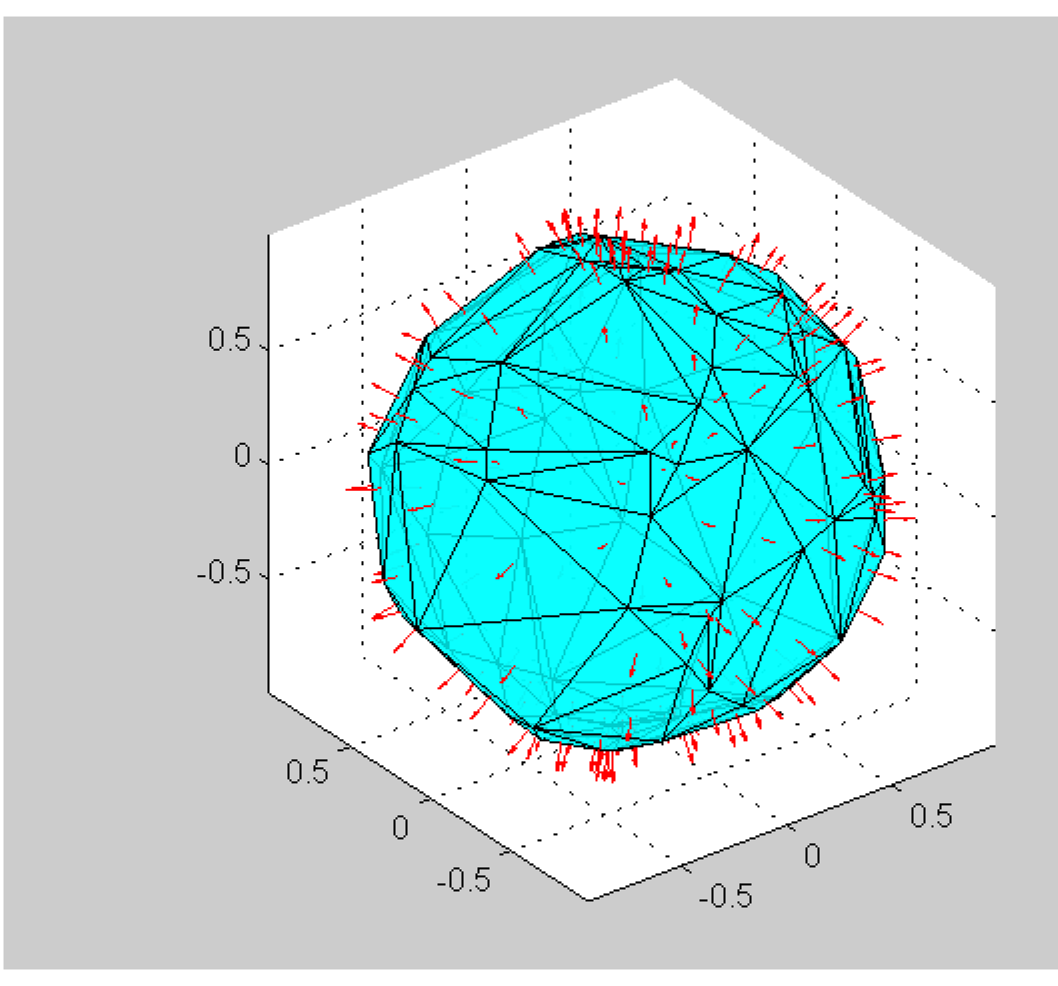
```
figure
trisurf(T,Xb(:,1),Xb(:,2),Xb(:,3), ...
     'FaceColor', 'cyan', 'faceAlpha', 0.8);
axis equal;
hold on;
```

Calulate the incenters and face normals.

```
P = incenter(TR);
fn = faceNormal(TR);
```

Display the normal vectors on the surface.

```
quiver3(P(:,1),P(:,2),P(:,3), ...
     fn(:,1),fn(:,2),fn(:,3),0.5, 'color','r');
hold off;
```



See Also `freeBoundary` | `delaunayTriangulation` |

Purpose

Triangulation sharp edges

Syntax

```
FE = featureEdges(TR,filterangle)
```

Description

`FE = featureEdges(TR,filterangle)` returns the feature edges in a 2-D triangulation.

Use this method to extract the sharp edges in the surface mesh for display purposes. A feature edge is an edge that has any of the following attributes:

- The edge is shared by only one triangle.
- The edge is shared by more than two triangles.
- The edge is shared by a pair of triangles with angular deviation greater than the `filterangle`.

Input Arguments

TR

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

filterangle

Filter angle, specified as a scalar value in the range $[0,\pi]$. `featureEdges` returns adjacent triangles that have a dihedral angle that deviates from π by an angle greater than `filterangle`.

Output Arguments

FE

Feature edge vertex IDs, returned as a two-column matrix. Each row of `FE` corresponds to a feature edge and contains two IDs:

- `FE(j,1)` is the ID of the vertex at the start of the edge.
- `FE(j,2)` is the ID of the vertex at end of the edge.

triangulation.featureEdges

Definitions

Vertex ID

A row number of the matrix, `TR.Points`. Use this ID to refer a specific vertex in the triangulation.

Examples

Feature Edges Shown on a Surface

Use `featureEdges` to find the feature edges of a surface and display them on a plot.

Create a surface triangulation.

```
x = [0 0 0 0 0 3 3 3 3 3 3 6 6 6 6 6 9 9 9 9 9 9]';
y = [0 2 4 6 8 0 1 3 5 7 8 0 2 4 6 8 0 1 3 5 7 8]';
DT = delaunayTriangulation(x,y);
T = DT(:,:);
```

Elevate the 2-D mesh to create a surface.

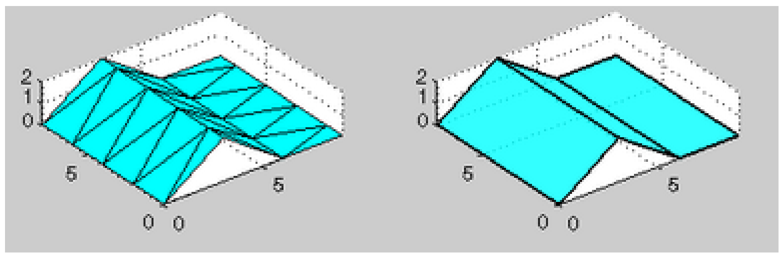
```
z = [0 0 0 0 0 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0]';
subplot(1,2,1);
trisurf(T,x,y,z,'FaceColor','cyan');
axis equal;
```

Compute the feature edges using a filter angle of `pi/6`.

```
TR = triangulation(T,x,y,z);
fe = featureEdges(TR,pi/6)';
subplot(1,2,2);
trisurf(TR,'FaceColor','cyan','EdgeColor',...
'none','FaceAlpha',0.8);
axis equal;
```

Add the feature edges to the plot.

```
hold on;
plot3(x(fe), y(fe), z(fe), 'k', 'LineWidth',1.5);
hold off;
```



See Also

edges | delaunayTriangulation |

triangulation.freeBoundary

Purpose

Triangulation facets referenced by only one triangle or tetrahedron

Syntax

```
FBtri = freeBoundary(TR)
[FBtri,FBpoints] = freeBoundary(TR)
```

Description

`FBtri = freeBoundary(TR)` returns the free boundary facets of `TR.ConnectivityList`. A facet in `TR` is on the free boundary if it is referenced by only one triangle or tetrahedron.

`[FBtri,FBpoints] = freeBoundary(TR)` also returns a matrix containing the vertices of the free boundary facets.

Input Arguments

TR

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

Output Arguments

FBtri

Triangulation connectivity list, returned as a matrix that contains the following information:

- Each row in `FBtri` represents a facet on the free boundary.
- Each element is a "Vertex ID" on page 1-4345.

The vertex IDs in `FBtri` reference a specific matrix, depending on the syntax you choose:

- If you call `freeBoundary` with one output argument, then the elements of `FBtri` are row numbers of `TR.Points`.
- If you call `freeBoundary` with two output arguments, then the elements of `FBtri` are row numbers of `FBpoints`.

FBpoints

Free boundary points, returned as a matrix containing the coordinates of the vertices of the free boundary facets. Each row of `FBpoints` contains coordinates of a vertex.

Definitions

Vertex ID

A row number of the matrix, `TR.Points` or `FBpoints`. Use this ID to refer a specific vertex in the triangulation.

Examples

Surface of 3-D Triangulation

Use `freeBoundary` to extract the facets of a 3-D triangulation that cover the surface of an object.

Load a 3-D triangulation.

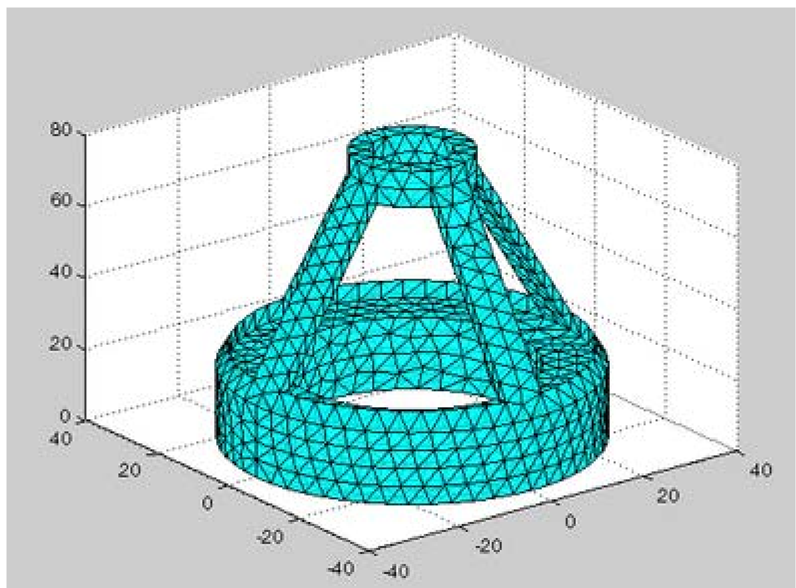
```
load tetmesh;
TR = triangulation(tet,X);
```

Compute the boundary triangulation.

```
[FBtri,FBpoints] = freeBoundary(TR);
```

Plot the boundary triangulation.

```
trisurf(FBtri,FBpoints(:,1),FBpoints(:,2),FBpoints(:,3), ...
       'FaceColor','cyan','FaceAlpha', 0.8);
```



Free Boundary of 2-D Delaunay Triangulation

Use `freeBoundary` when you want to highlight the outer edges of a 2-D Delaunay triangulation.

Create a triangulation from a random set of points.

```
x = gallery('uniformdata',[20,1],0);
y = gallery('uniformdata',[20,1],1);
DT = delaunayTriangulation(x,y);
```

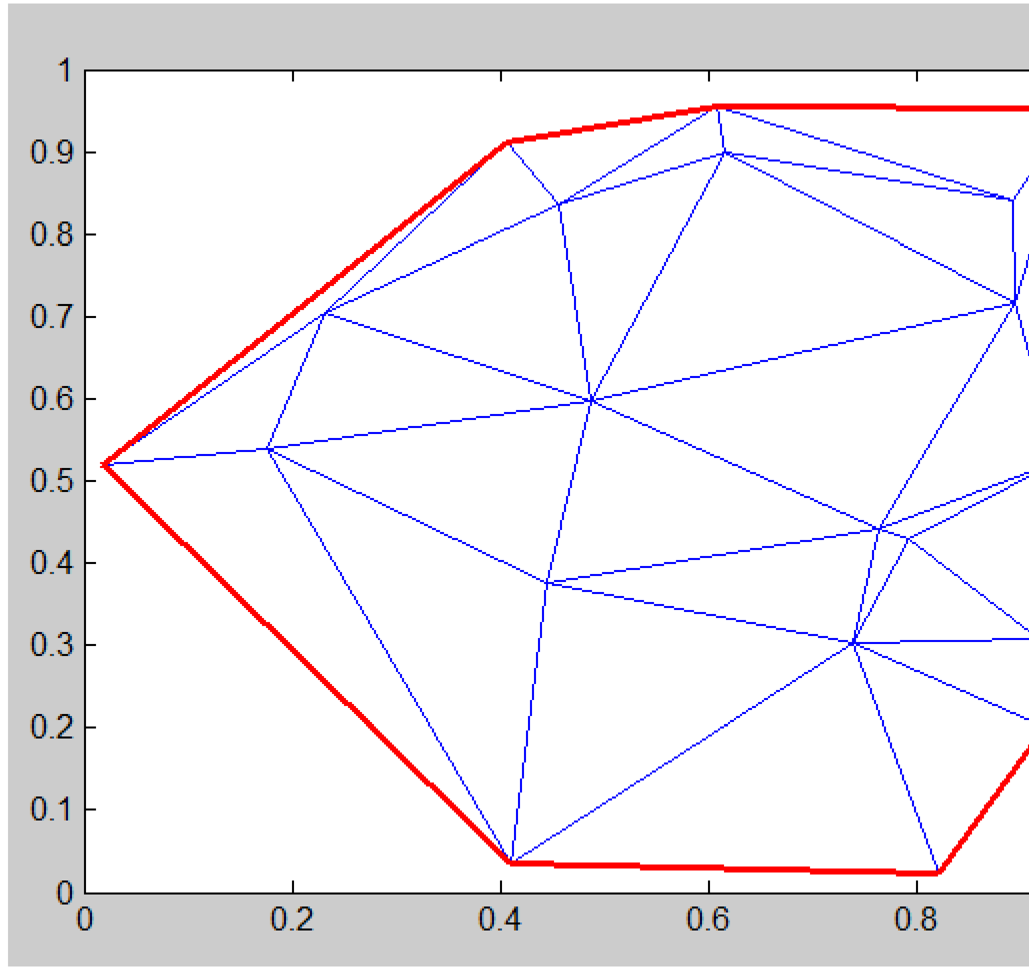
Find the free boundary edges.

```
fe = freeBoundary(DT)';
```

Plot the mesh and highlight the free boundary edges in red.

```
triplot(DT);
hold on;
```

```
plot(x(fe),y(fe),'-r','LineWidth',2) ;
hold off;
```



See Also

featureEdges | faceNormal | delaunayTriangulation |

triangulation.incenter

Purpose

Incenter of triangle or tetrahedron

Syntax

```
IC = incenter(TR,ti)
[IC,r] = incenter(TR,ti)
```

Description

`IC = incenter(TR,ti)` returns the coordinates of the incenter of each triangle or tetrahedron specified by `ti`.

`[IC,r] = incenter(TR,ti)` also returns the radii of the inscribed circles or spheres.

Input Arguments

TR

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

ti

Triangle or tetrahedron IDs, specified as a column vector.

Output Arguments

IC

Incenters, returned as a matrix. Each row of `IC` contains the coordinates of an incenter. For example, `IC(j,:)` is the incenter of `ti(j)`.

r

Radii of the inscribed circles or spheres, returned as a vector. `r(j)` is the radius of the inscribed circle or sphere whose center is `IC(j,:)`.

Definitions

Triangle or Tetrahedron ID

A row number of the matrix, `TR.ConnectivityList`. Use this ID to refer a specific triangle or tetrahedron.

Examples

Find Incenters in 3-D Triangulation

Load a 3-D triangulation.

```
load tetmesh
```

Calculate the incenters of the first five tetrahedra.

```
TR = triangulation(tet,X);
IC = incenter(TR,[1:5]')

IC =

   -6.1083  -31.0234    8.1439
   -2.1439  -31.0283    5.8742
   -1.9555  -31.9463    7.4112
   -4.3019  -30.8460   10.5169
   -3.1596  -29.3642    6.1851
```

Find Incenters in 2-D Delaunay Triangulation

Create the Delaunay triangulation.

```
x = [0 1 1 0 0.5]';
y = [0 0 1 1 0.5]';
DT = delaunayTriangulation(x,y);
```

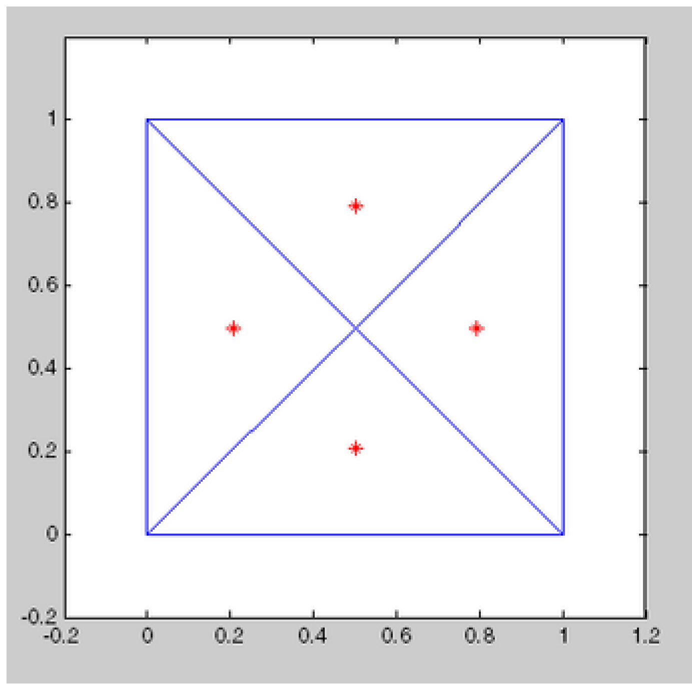
Calculate incenters of the triangles

```
IC = incenter(DT)

IC =

    0.2071    0.5000
    0.5000    0.7929
    0.7929    0.5000
    0.5000    0.2071
```

Plot the triangles and incenters.

```
triplot(DT);
axis equal;
```

```
axis([-0.2 1.2 -0.2 1.2]);
hold on;
plot(IC(:,1),IC(:,2),'*r');
hold off;
```



See Also

`circumcenter` | `delaunayTriangulation` |

Purpose

Test if two vertices are connected by edge

Syntax

```
tf = isConnected(TR,vstart,vend)
tf = isConnected(TR,E)
```

Description

`tf = isConnected(TR,vstart,vend)` returns a `logical` array of `true` or `false` values that indicate whether the specified pairs of vertices are connected by an edge. Element `tf(j)` is `true` when the vertices, `vstart(j)` and `vend(j)`, are connected by an edge.

`tf = isConnected(TR,E)` specifies the edge start and end indices in matrix `E`. Element `tf(j)` is `true` when the vertices, `E(j,1)` and `E(j,2)`, are connected by an edge.

Input Arguments

TR

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

vstart

IDs of start vertices, specified as a column vector. Each vertex ID refers to a vertex at the start of an edge.

vend

IDs of end vertices, specified as a column vector. Each vertex ID refers to a vertex at the end of an edge.

E

IDs of the edge vertices, specified as a two-column matrix. Each row of `E` corresponds to a candidate edge and contains two IDs:

- `E(j,1)` is the ID of the vertex at the start of a candidate edge.
- `E(j,2)` is the ID of the vertex at end of the edge.

Output Arguments

tf

Logical values, returned as a column vector. Element `tf(j)` is `true` when either of the following are true:

- The vertices, `vstart(j)` and `vend(j)`, are connected by an edge.
- The vertices, `E(j,1)` and `E(j,2)`, are connected by an edge.

Otherwise, `tf(j)` is false.

Definitions

Vertex ID

A row number of the matrix, `TR.Points`. Use this ID to refer a specific vertex in the triangulation.

Examples

Determine Whether Vertices are connected by an Edge in 2-D

Load a 2-D triangulation.

```
load trimesh2d
TR = triangulation(tri,x,y);
```

Determine whether vertices `3` and `117` are connected by an edge.

```
isConnected(TR,3,117)
```

```
ans =

     1
```

The vertices are connected by an edge.

Determine whether vertices `3` and `164` are connected by an edge.

```
isConnected(TR,3,164)
```

```
ans =
```

```
     0
```

The vertices are not connected by an edge.

Determine Whether Vertices are connected by an Edge in 3-D

```
X = gallery('uniformdata',[10,3],0);
DT = delaunayTriangulation(X);
```

Determine whether vertices 2 and 7 are connected by an edge.

```
E = [2 7];
isConnected(DT,E)
```

```
ans =

     1
```

The vertices are connected by an edge.

See Also

edges | edgeAttachments | delaunayTriangulation |

triangulation.neighbors

Purpose

Neighbors to specified triangle or tetrahedron

Syntax

```
N = neighbors(TR,ti)
N = neighbors(TR)
```

Description

`N = neighbors(TR,ti)` returns the neighbors of the triangles or tetrahedra specified in `ti`.

`N = neighbors(TR)` returns the neighbors of all the triangles or tetrahedra.

Input Arguments

TR

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

ti

Triangle or tetrahedron IDs, specified as a column vector.

Output Arguments

TN

IDs of the neighboring triangles or tetrahedra, returned as a matrix. The elements in `TN(i,:)` are the neighbors associated with `ti(i)`.

By convention, `TN(i,j)` is the neighbor opposite the `j`th vertex of `ti(i)`. If a triangle or tetrahedron has one or more boundary facets, the nonexistent neighbors are represented as `NaN` values in `TN`.

Definitions

Triangle or Tetrahedron ID

A row number of the matrix, `TR.ConnectivityList`. Use this ID to refer a specific triangle or tetrahedron.

Vertex ID

A row number of the matrix, `TR.Points`. Use this ID to refer a specific vertex in the triangulation.

Examples

Query Neighbors in 2-D Delaunay Triangulation

Create a Delaunay triangulation from a set of random points.

```
x = gallery('uniformdata',[10,1],0);
y = gallery('uniformdata',[10,1],1);
DT = delaunayTriangulation(x,y);
```

Find the neighbors of the first triangle.

```
TN = neighbors(DT,1)

TN =

   NaN     4     5
```

The first triangle has a boundary edge and two neighbors.

Examine the vertex IDs of the first neighbor, `TN(2)`.

```
DT.ConnectivityList(TN(2),:)

ans =

     2     4     7
```

Query Neighbors in 3-D Triangulation

Create the Delaunay triangulation.

```
load tetmesh
TR = triangulation(tet,X);
```

Find the neighbors to each triangle in the triangulation.

```
TN = neighbors(TR);
```

Find the neighbors of the fifth tetrahedron.

```
TN(5,:)
```

```
ans =

        2360        1539           2        1851
```

Examine the vertex IDs of the first neighbor, `TN(1)`.

```
TR.ConnectivityList(TN(1),:)
```

```
ans =

        1093         891         893         858
```

See Also

`edgeAttachments` | `delaunayTriangulation` |

Purpose

Size of triangulation connectivity list

Syntax

```
sz = size(TR)
```

Description

`sz = size(TR)` returns the size the triangulation connectivity list.

Input Arguments

TR

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

Output Arguments

sz

Size of connectivity list, returned as a two-element row vector. The first element, `sz(1)` is the number of triangles or tetrahedra in the triangulation, and `sz(2)` is the number of vertices per triangle or tetrahedron.

Examples

Size of 2-D Triangulation

Create a triangulation.

```
P = [ 2.5    8.0
      6.5    8.0
      2.5    5.0
      6.5    5.0
      1.0    6.5
      8.0    6.5];

T = [5  3  1;
     3  2  1;
     3  4  2;
     4  6  2];

TR = triangulation(T,P);
```

Get the size of the connectivity list.

triangulation.size

```
size(TR)

ans =

     4     3
```

The triangulation has 4 triangles, and each triangle has 3 vertices.

See Also

`sizedelaunayTriangulation` |

triangulation.vertexAttachments

Purpose

Triangles or tetrahedra attached to specified vertex

Syntax

```
ti = vertexAttachments(TR,vi)
ti = vertexAttachments(TR)
```

Description

`ti = vertexAttachments(TR,vi)` returns the triangles or tetrahedra attached to the specified vertices, `vi`.

`ti = vertexAttachments(TR)` returns the triangles or tetrahedra attached to every vertex in the triangulation.

Input Arguments

TR

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

vi

Query vertices, specified as a column vector of vertex IDs.

Output Arguments

ti

IDs of the triangles or tetrahedra attached to the vertices, returned as an `m`-by-`1` cell array. Each cell in `ti` contains the IDs of the attached triangles or tetrahedra. `ti{j}` contains the attachments to the vertex, `vi(j)`. The attachments are returned in a cell array because the number of triangles or tetrahedra associated with each vertex can vary.

Definitions

Triangle or Tetrahedron ID

A row number of the matrix, `TR.ConnectivityList`. Use this ID to refer a specific triangle or tetrahedron.

Vertex ID

A row number of the matrix, `TR.Points`. Use this ID to refer a specific vertex in the triangulation.

Examples

Attachments to Specific Vertex in 2-D Delaunay Triangulation

Locate and plot the attachments to a specific vertex.

Create a Delaunay triangulation from a set of random points.

```
x = gallery('uniformdata',[20,1],0);
y = gallery('uniformdata',[20,1],1);
DT = delaunayTriangulation(x,y);
```

Find the triangles attached to the fifth vertex.

```
ti = vertexAttachments(DT,5);
ti{:}
```

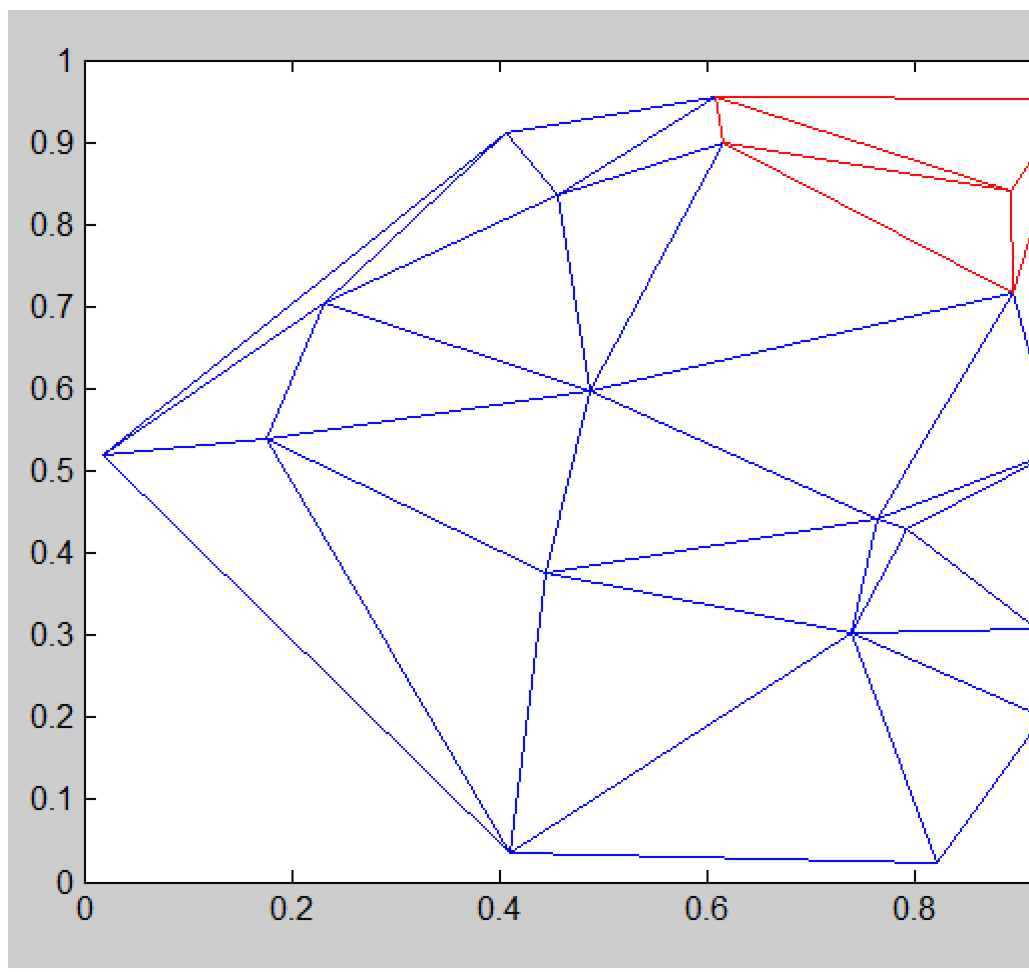
```
ans =

    18    23    21    22
```

Plot the triangulation.

```
triplot(DT);
hold on;
```

Plot the triangles attached to vertex 5 (in red).

```
triplot(DT(ti{:},:),x,y,'Color','r');
hold off;
```



See Also

`edgeAttachments` | `delaunayTriangulation` |

triangulation.vertexNormal

Purpose Triangulation vertex normal

Syntax

```
VN = vertexNormal(TR,vi)
VN = vertexNormal(TR)
```

Description

`VN = vertexNormal(TR,vi)` returns the unit normal vector to each of the specified vertices in `vi`.

`VN = vertexNormal(TR)` returns the normal information for all vertices in the triangulation.

Input Arguments

TR

Triangulation representation, see `triangulation` or `delaunayTriangulation`.

vi

IDs of vertices to query, specified as a column vector. Each element in `vi` is a “Vertex ID” on page 1-4362.

Output Arguments

VN

Vertex normals, returned a matrix. Each row, `VN(j,:)`, is the unit normal vector at the vertex `vi(j)`. The vector at `VN(j,:)` is the average unit normal of the faces attached to vertex `vi(j)`.

If you do not specify `vi`, then `vertexNormal` returns the unit normal information for all vertices in the triangulation. In this case, the normal associated with `TR.Points(j,:)` is `VN(j,:)`.

Definitions

Vertex ID

A row number of the matrix, `TR.Points`. Use this ID to refer a specific vertex in the triangulation.

Examples

Unit Normal Vectors to the Surface of a Cube

Create a 3-D triangulation representing the volume of a cube.

```
[X,Y,Z] = meshgrid(1:4);
X = X(:);
Y = Y(:);
Z = Z(:);
dt = delaunayTriangulation(X,Y,Z);
```

Find the surfaces of the cube and isolate them in a 2-D triangulation.

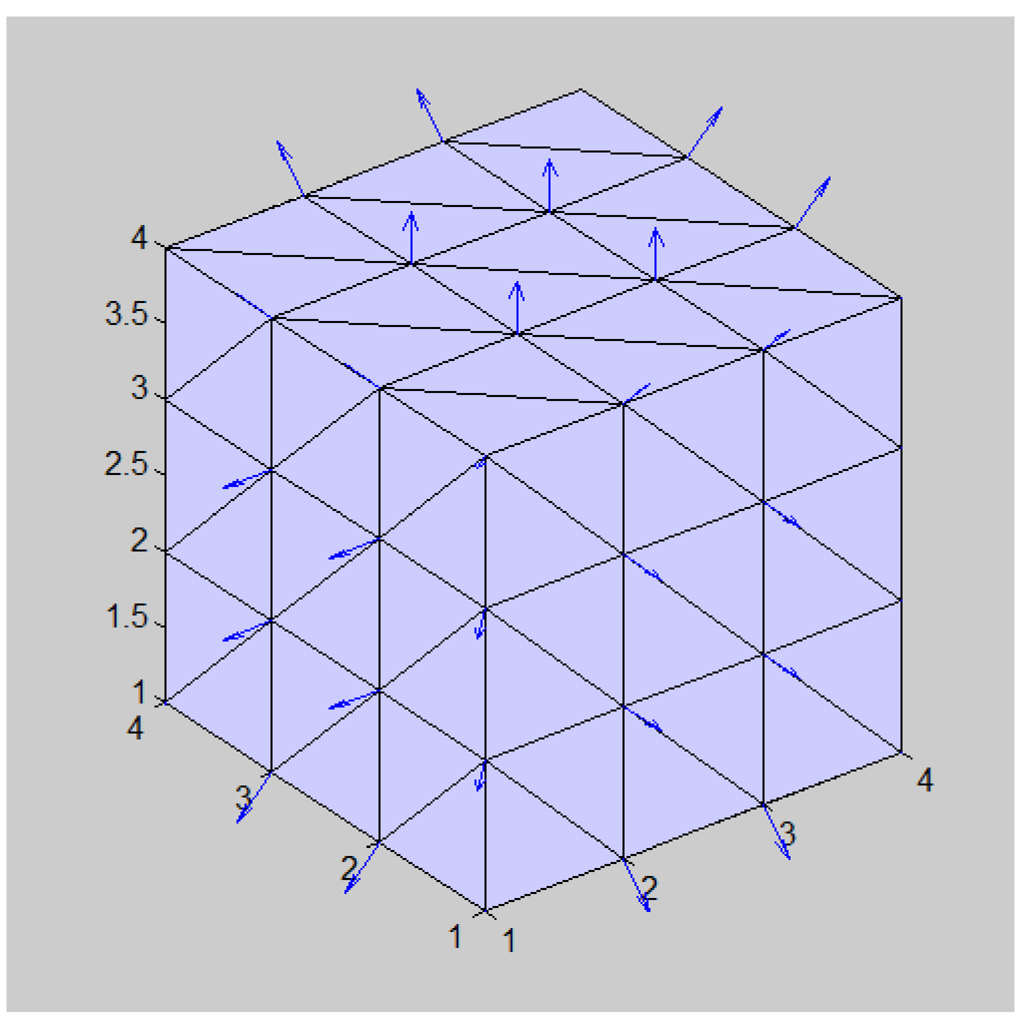
```
[Tfb,Xfb] = freeBoundary(dt);
TR = triangulation(Tfb,Xfb);
```

Find the unit normal vectors to the triangles on the surface of the cube.

```
vn = vertexNormal(TR);
```

Plot the results.

```
trisurf(TR,'FaceColor', [0.8 0.8 1.0]);
axis equal
hold on
quiver3(Xfb(:,1),Xfb(:,2),Xfb(:,3),...
     vn(:,1),vn(:,2),vn(:,3),0.5,'color','b');
hold off
```



See Also `freeBoundary` | `delaunayTriangulation` |

Purpose

Real part of complex number

Syntax

```
X = real(Z)
```

Description

`X = real(Z)` returns the real part of the elements of the complex array `Z`.

Examples

`real(2+3*i)` is `2`.

See Also

`abs` | `angle` | `conj` | `i` | `j` | `imag`

reallog

Purpose

Natural logarithm for nonnegative real arrays

Syntax

```
Y = reallog(X)
```

Description

`Y = reallog(X)` returns the natural logarithm of each element in array `X`. Array `X` must contain only nonnegative real numbers. The size of `Y` is the same as the size of `X`.

Examples

```
M = magic(4)

M =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

reallog(M)

ans =
    2.7726    0.6931    1.0986    2.5649
    1.6094    2.3979    2.3026    2.0794
    2.1972    1.9459    1.7918    2.4849
    1.3863    2.6391    2.7081         0
```

See Also

`log` | `realpow` | `realsqrt`

Purpose

Largest positive floating-point number

Syntax

```
n = realmax
```

Description

`n = realmax` returns the largest finite floating-point number in IEEE double precision.

`realmax('double')` is the same as `realmax` with no arguments.

`realmax('single')` returns the largest finite floating-point number in IEEE single precision.

Examples

Find the value of the constant `realmax`:

```
ndouble = realmax
nsingle = realmax('single')
```

```
ndouble =

  1.7977e+308


nsingle =

  3.4028e+38
```

See Also

`eps` | `realmin` | `intmax`

Tutorials

- “Floating-Point Numbers”

realmin

Purpose

Smallest positive normalized floating-point number

Syntax

```
n = realmin
realmin('double')
realmin('single')
```

Description

`n = realmin` returns the smallest positive normalized floating-point number in IEEE double precision.

`realmin('double')` is the same as `realmin` with no arguments.

`realmin('single')` returns the smallest positive normalized floating-point number in IEEE single precision.

Examples

Find the value of the constant `realmin`:

```
ndouble = realmin
nsingle = realmin('single')
```

```
ndouble =

  2.2251e-308


nsingle =

  1.1755e-38
```

See Also

`eps` | `realmax` | `intmin`

Tutorials

- "Floating-Point Numbers"

Purpose Array power for real-only output

Syntax

```
Z = realpow(X,Y)
```

Description

`Z = realpow(X,Y)` raises each element of array `X` to the power of its corresponding element in array `Y`. Arrays `X` and `Y` must be the same size. The range of `realpow` is the set of all real numbers, i.e., all elements of the output array `Z` must be real.

Examples

```
X = -2*ones(3,3)

X =
     -2     -2     -2
     -2     -2     -2
     -2     -2     -2

Y = pascal(3)

ans =
     1     1     1
     1     2     3
     1     3     6

realpow(X,Y)

ans =
    -2    -2    -2
    -2     4    -8
    -2    -8    64
```

See Also

`reallog` | `realsqrt` | `Arithmetic Operator .^`

realsqrt

Purpose

Square root for nonnegative real arrays

Syntax

```
Y = realsqrt(X)
```

Description

`Y = realsqrt(X)` returns the square root of each element of array `X`. Array `X` must contain only nonnegative real numbers. The size of `Y` is the same as the size of `X`.

Examples

```
M = magic(4)

M =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

realsqrt(M)

ans =
    4.0000    1.4142    1.7321    3.6056
    2.2361    3.3166    3.1623    2.8284
    3.0000    2.6458    2.4495    3.4641
    2.0000    3.7417    3.8730    1.0000
```

See Also

`reallog` | `realpow` | `sqrt` | `sqrtm`

Purpose

Record data and event information to file

Syntax

```
record(obj)
record(obj,'switch')
```

Description

`record(obj)` toggles the recording state for the serial port object, `obj`.

`record(obj,'switch')` initiates or terminates recording for `obj`. *`switch`* can be `on` or `off`. If *`switch`* is `on`, recording is initiated. If *`switch`* is `off`, recording is terminated.

Tips

Before you can record information to disk, `obj` must be connected to the device with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to record information while `obj` is not connected to the device. Each serial port object must record information to a separate file. Recording is automatically terminated when `obj` is disconnected from the device with `fclose`.

The `RecordName` and `RecordMode` properties are read-only while `obj` is recording, and must be configured before using `record`.

For a detailed description of the record file format and the properties associated with recording data and event information to a file, refer to Debugging: Recording Information to Disk.

Examples

This example creates the serial port object `s` on a Windows platform. It connects `s` to the device, configures `s` to record information to a file, writes and reads text data, and then disconnects `s` from the device.

```
s = serial('COM1');
fopen(s)
s.RecordDetail = 'verbose';
s.RecordName = 'MySerialFile.txt';
record(s,'on')
fprintf(s,'*IDN?')
out = fscanf(s);
record(s,'off')
```

```
fclose(s)
```

See Also

fclose | fopen | RecordDetail | RecordMode | RecordName | RecordStatus | Status

Purpose

Record audio to `audiorecorder` object

Syntax

```
record(recorderObj)
record(recorderObj, length)
```

Description

`record(`*`recorderObj`*`)` records audio from an input device, such as a microphone connected to your system. *`recorderObj`* is an `audiorecorder` object that defines the sample rate, bit depth, and other properties of the recording.

`record(`*`recorderObj`*`, `*`length`*`)` records for the number of seconds specified by *`length`*.

Examples

Record 5 seconds of your speech with a microphone:

```
myVoice = audiorecorder;

% Define callbacks to show when
% recording starts and completes.
myVoice.StartFcn = 'disp(''Start speaking.'')';
myVoice.StopFcn = 'disp(''End of recording.'')';

record(myVoice, 5);
```

To listen to the recording, call the `play` method:

```
play(myVoice);
```

See Also

`audiorecorder` | `getaudiodata` | `recordblocking`

How To

- “Record Audio”
- “Recording or Playing Audio within a Function”

audiorecorder.recordblocking

Purpose

Record audio to `audiorecorder` object, holding control until recording completes

Syntax

`recordblocking(*recorderObj*, *length*)`

Description

`recordblocking(*recorderObj*, *length*)` records audio from an input device, such as a microphone connected to your system, for the number of seconds specified by *`length`*. The `recordblocking` method does not return control until recording completes. *`recorderObj`* is an `audiorecorder` object that defines the sample rate, bit depth, and other properties of the recording.

Examples

Record 5 seconds of your speech with a microphone, and play it back:

```
myVoice = audiorecorder;

disp('Start speaking.');
recordblocking(myVoice, 5);
disp('End of recording. Playing back ...');

play(myVoice);
```

See Also

`audiorecorder` | `getaudiodata` | `record`

How To

- "Record Audio"

Purpose

Create 2-D rectangle object

Syntax

```
rectangle
rectangle('Position',[x,y,w,h])
rectangle('Curvature',[x,y])
rectangle('PropertyName',propertyvalue,...)
h = rectangle(...)
```

Properties

For a list of properties, see Rectangle Properties.

Description

`rectangle` draws a rectangle with `Position [0,0,1,1]` and `Curvature [0,0]` (i.e., no curvature).

`rectangle('Position',[x,y,w,h])` draws the rectangle from the point `x`,`y` and having a width of `w` and a height of `h`. Specify values in axes data units.

Note that, to display a rectangle in the specified proportions, you need to set the axes data aspect ratio so that one unit is of equal length along both the x and y axes. You can do this with the command `axis equal` or `daspect([1,1,1])`.

`rectangle('Curvature',[x,y])` specifies the curvature of the rectangle sides, enabling it to vary from a rectangle to an ellipse. The horizontal curvature `x` is the fraction of width of the rectangle that is curved along the top and bottom edges. The vertical curvature `y` is the fraction of the height of the rectangle that is curved along the left and right edges.

The values of `x` and `y` can range from `0` (no curvature) to `1` (maximum curvature). A value of `[0,0]` creates a rectangle with square sides. A value of `[1,1]` creates an ellipse. If you specify only one value for `Curvature`, then the same length (in axes data units) is curved along both horizontal and vertical sides. The amount of curvature is determined by the shorter dimension.

`rectangle('PropertyName',propertyvalue,...)` draws the rectangle using the values for the property name/property value pairs specified and default values for all other properties. For a description of the properties, see Rectangle Properties.

`h = rectangle(...)` returns the handle of the rectangle object created.

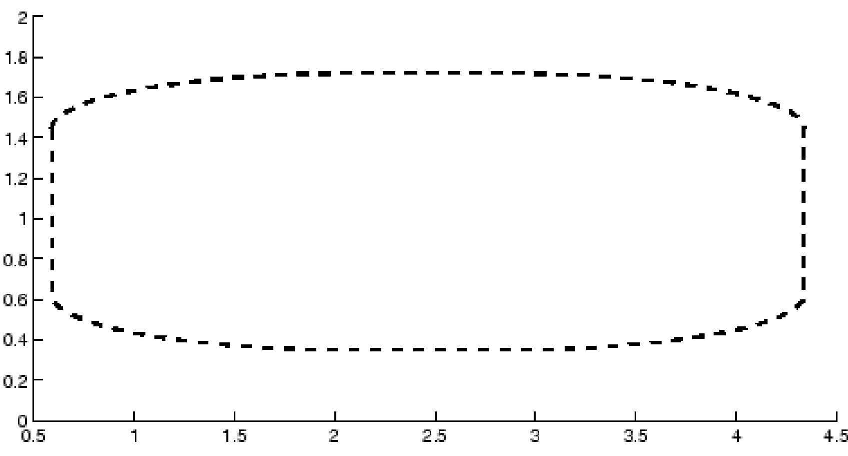
Tips

Rectangle objects are 2-D and can be drawn in an axes only if the view is `[0 90]` (i.e., `view(2)`). Rectangles are children of axes and are defined in coordinates of the axes data.

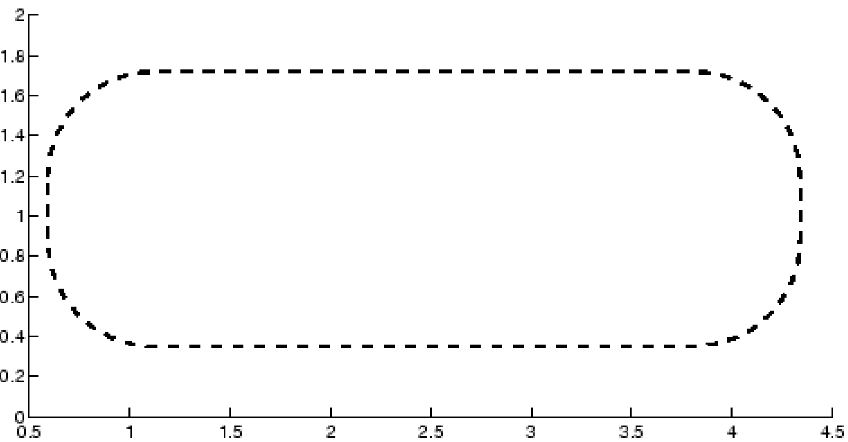
Examples

This example sets the data aspect ratio to `[1,1,1]` so that the rectangle is displayed in the specified proportions (`daspect`). Note that the horizontal and vertical curvature can be different. Also, note the effects of using a single value for `Curvature`.

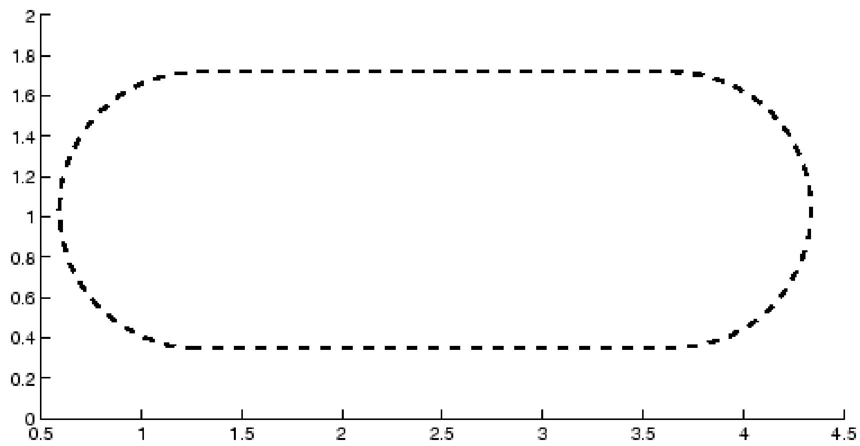
```
rectangle('Position',[0.59,0.35,3.75,1.37],...
          'Curvature',[0.8,0.4],...
         'LineWidth',2,'LineStyle','--')
daspect([1,1,1])
```



Specifying a single value of `[0.4]` for `Curvature` produces



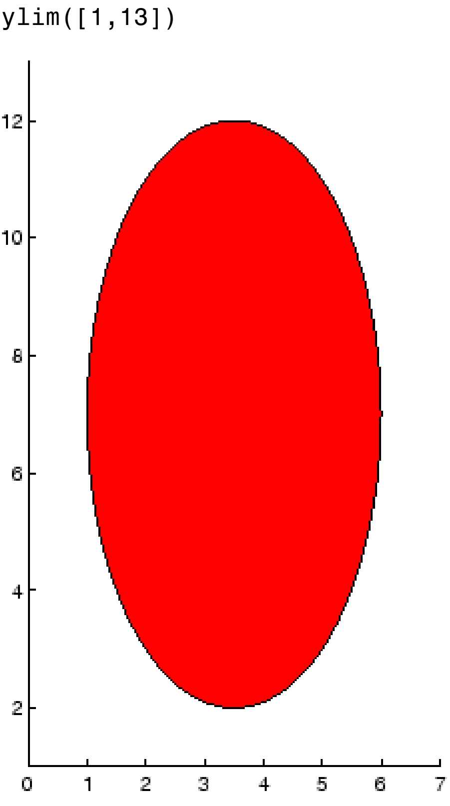
A `Curvature` of `[1]` produces a rectangle with the shortest side completely round:



This example creates an ellipse and colors the face red.

```
rectangle('Position',[1,2,5,10],'Curvature',[1,1],...
          'FaceColor','r')
daspect([1,1,1])
xlim([0,7])
```

```
ylim([1,13])
```



Setting Default Properties

You can set default rectangle properties on the axes, figure, and `root object` levels:

```
set(0,'DefaultRectangleProperty',PropertyValue...)
set(gcf,'DefaultRectangleProperty',PropertyValue...)
set(gca,'DefaultRectangleProperty',PropertyValue...)
```

where *`Property`* is the name of the rectangle property whose default value you want to set and `PropertyValue` is the value you are specifying. Use `set` and `get` to access the surface properties.

See Also

`line` | `patch` | `annotation` | Rectangle Properties

Rectangle Properties

Purpose

Define rectangle properties

Creating Rectangle Objects

Use `rectangle` to create rectangle objects.

Modifying Properties

You can set and query graphics object properties in two ways:

- "The Property Editor" is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see "Setting Default Property Values".

See "Core Graphics Objects" for general information about this type of object.

Rectangle Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`Annotation`

`hg.Annotation` object (read-only)

Handle of `Annotation` *object.* The `Annotation` property enables you to specify whether this rectangle object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the rectangle object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Represent this rectangle object in a legend (default)
`off`	Do not include this rectangle object in a legend
`children`	Same as `on` because rectangle objects do not have children

Setting the IconDisplayStyle property

Set the `IconDisplayStyle` of a graphics object with handle `hobj` to `off`:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Using the IconDisplayStyle property

See "Controlling Legends" for more information and examples.

`BeingDeleted`

`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Button press callback function. Executes whenever you press a mouse button while the pointer is over the rectangle object.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

Set this property to a function handle that references the callback. The function must define at least two input arguments (handle of object associated with the button down event and an event structure, which is empty for this property).

The following example shows how to access the callback object's handle as well as the handle of the figure that contains the object from the callback function.

```
function button_down(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   sel_typ = get(gcbf,'SelectionType')
   switch sel_typ
      case 'normal'
         disp('User clicked left-mouse button')
         set(src,'Selected','on')
      case 'extend'
         disp('User did a shift-click')
         set(src,'Selected','on')
      case 'alt'
         disp('User did a control-click')
         set(src,'Selected','on')
         set(src,'SelectionHighlight','off')
   end
end
```

Suppose `h` is the handle of a rectangle object and the `button_down` function is on your MATLAB path. The following statement assigns the `button_down` function to the `ButtonDownFcn` property:

```
set(h,'ButtonDownFcn',@button_down)
```

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`Children`
: vector of handles

  The empty matrix; rectangle objects have no children.

`Clipping`
: `{on} | off`

Clipping mode. MATLAB clips rectangles to the axes plot box by default. If you set `Clipping` to `off`, rectangles are displayed outside the axes plot box. This can occur if you create a rectangle, set `hold` to `on`, freeze axis scaling (`axis` set to `manual`), and then create a larger rectangle.

`CreateFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback function executed during object creation. Executes when MATLAB creates a rectangle object. You must define this property as a default value for rectangles or in a call to the `rectangle` function to create a new rectangle object. For example, the statement:

```
set(0,'DefaultRectangleCreateFcn',@rect_create)
```

defines a default value for the rectangle `CreateFcn` property on the root level that sets the axes `DataAspectRatio` whenever you create a rectangle object. The callback function must be on your MATLAB path when you execute the above statement.

```
function rect_create(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
  axh = get(src,'Parent');
  set(axh,'DataAspectRatio',[1,1,1]))
end
```

MATLAB executes this function after setting all rectangle properties. Setting this property on an existing rectangle object has no effect. The function must define at least two input arguments (handle of object created and an event structure, which is empty for this property).

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function

and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`Curvature`

one- or two-element vector `[x,y]`

Amount of horizontal and vertical curvature. Specifies the curvature of the rectangle sides, which enables the shape of the rectangle to vary from rectangular to ellipsoidal. The horizontal curvature `x` is the fraction of width of the rectangle that is curved along the top and bottom edges. The vertical curvature `y` is the fraction of the height of the rectangle that is curved along the left and right edges.

The values of `x` and `y` can range from `0` (no curvature) to `1` (maximum curvature). A value of `[0,0]` creates a rectangle with square sides. A value of `[1,1]` creates an ellipse. If you specify only one value for `Curvature`, then the same length (in axes data units) is curved along both horizontal and vertical sides. The amount of curvature is determined by the shorter dimension.

`DeleteFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Delete rectangle callback function. Executes when you delete the rectangle object (for example, when you issue a `delete` command or clear the axes `cla` or figure `clf`).

For example, the following function displays object property data before the object is deleted.

```
function delete_fcn(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
```

```
   obj_tp = get(src,'Type');
   disp([obj_tp, ' object deleted'])
   disp('Its user data is:')
   disp(get(src,'UserData'))
end
```

MATLAB executes the function before deleting the object's properties so these values are available to the callback function. The function must define at least two input arguments (handle of object being deleted and an event structure, which is empty for this property).

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DisplayName`
string

String used by legend. The `legend` function uses the `DisplayName` property to label the rectangle object in the legend. The default is an empty string.

- If you specify string arguments with the `legend` function, MATLAB set `DisplayName` to the corresponding string and uses that string for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, MATLAB set `DisplayName` to the edited string.

- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add a legend programmatically that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more information and examples.

`EdgeColor`
`{ColorSpec} | none`

Color of the rectangle edges. Specifies the color of the rectangle edges as a color or specifies that no edges be drawn.

`EraseMode`
`{normal} | none | xor | background`

Erase mode. Controls the technique MATLAB uses to draw and erase rectangle objects. Use alternative erase modes for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the rectangle when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it because MATLAB stores no information about its former location.
- `xor` — Draw and erase the rectangle by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath the rectangle.

However, the rectangle's color depends on the color of whatever is beneath it on the display.

- `background` — Erase the rectangle by drawing it in the axes background `Color`, or the figure background `Color` if the axes `Color` is `none`. This damages objects that are behind the erased rectangle, but rectangles are always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors (for example, performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`FaceColor`
`ColorSpec | {none}`

Color of rectangle face. Specifies the color of the rectangle face, which is not colored by default.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

- `on` — Handles are always visible.

- `callback` — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

You can set the Root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

`HitTest`

`{on} | off`

Selectable by mouse click. Determines if the rectangle can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the

rectangle. If `HitTest` is `off`, clicking the rectangle selects the object below it (which might be the axes containing it).

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the `Interruptible` property affects only the callbacks for the`ButtonDownFcn` property. A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both the callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see “Control Callback Execution and Interruption”.

`LineStyle`

`{-} | -- | : | -. | none`

Line style of rectangle edge.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line
'none'	No line

`LineWidth`

size in points

Width of the rectangle edge line. Specify this value in points. 1 point = $^1/_{72}$ inch. The default value is `0.5` points.

`Parent`
handle of axes, hggroup, or hgtransform

Parent of rectangle object. Contains the handle of the rectangle object's parent. The parent of a rectangle object is the axes, hggroup, or hgtransform object that contains it.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Position`
four-element vector `[x,y,width,height]`

Location and size of rectangle. Specifies the location and size of the rectangle in the data units of the axes. The point defined by `x`, `y` specifies one corner of the rectangle, and `width` and `height` define the size in units along the *x*- and *y*-axes respectively.

`Selected`
`on | off`

Is object selected? When this property is `on` MATLAB displays selection handles if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing handles at each vertex. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`Tag`
string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

`Type`
string (read-only)

Class of graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For rectangle objects, `Type` is always '`rectangle`'.

`UIContextMenu`
handle of uicontextmenu object

Associate a context menu with the rectangle. Assign this property the handle of a uicontextmenu object created in the same figure as the rectangle. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the rectangle.

`UserData`
matrix

User-specified data. Data you want to associate with the rectangle object. The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`
`{on} | off`

Rectangle visibility. By default, all rectangles are visible. When set to `off`, the rectangle is not visible, but still exists, and you can `get` and `set` its properties.

See Also

`rectangle`

Purpose

Rectangle intersection area

Syntax

```
area = rectint(A,B)
```

Description

`area = rectint(A,B)` returns the area of intersection of the rectangles specified by position vectors `A` and `B`.

If `A` and `B` each specify one rectangle, the output `area` is a scalar.

`A` and `B` can also be matrices, where each row is a position vector. `area` is then a matrix giving the intersection of all rectangles specified by `A` with all the rectangles specified by `B`. That is, if `A` is `n`-by-`4` and `B` is `m`-by-`4`, then `area` is an `n`-by-`m` matrix where `area(i,j)` is the intersection area of the rectangles specified by the `i`th row of `A` and the `j`th row of `B`.

Note A position vector is a four-element vector `[x,y,width,height]`, where the point defined by `x` and `y` specifies one corner of the rectangle, and `width` and `height` define the size in units along the x and y axes respectively.

See Also

`polyarea`

Purpose

Set option to move deleted files to recycle folder

Syntax

```
recycle
stat = recycle
previousStat = recycle state
previousStat = recycle('state')
```

Description

`recycle` displays the current state, `on` or `off`, for recycling files you remove using the `delete` function. When the value is `on`, deleted files move to a different location. The location varies by platform—see "Deleting Files and Folders Using Functions". When the value is `off`, the `delete` function permanently removes the files. For details, see the Tips section.

`stat = recycle` returns the current state for recycling files to the character array `stat`.

`previousStat = recycle` *`state`* sets the recycle option for MATLAB to the specified state, either `on` or `off`. The `previousStat` value is the recycle state before running the statement.

`previousStat = recycle('state')` is the function form of the syntax.

Tips

The preference for **Deleting files** sets the state of the `recycle` function at startup. When you change the preference, it changes the state of `recycle`. When you change the state of `recycle`, it does not change the preference. Use `recycle` to override the behavior of the preference. For example, regardless of the setting for the **Deleting files** preference, to remove `thisfile.m` permanently, run:

```
recycle('off')
delete('thisfile.m')
```

After setting the `recycle` state to `off`, all files you delete using the `delete` function are deleted permanently until you do one of the following:

- Run `recycle('on')`

- Restart MATLAB. Upon startup, MATLAB sets the state for `recycle` to match the **Deleting files** preference.

Examples

Start from a state where file recycling is off. Verify the current recycle state:

```
recycle
ans =
   off
```

Turn file recycling on. Delete a file and move it to the recycle bin or temporary folder:

```
recycle on;
delete myfile.txt
```

See Also

`delete` | `dir` | `ls` | `rmdir`

How To

- “General Preferences”

Purpose

Reduce number of patch faces

Syntax

```
nfv = reducepatch(p,r)
nfv = reducepatch(fv,r)
nfv = reducepatch(p) or nfv = reducepatch(fv)
reducepatch(...,'fast')
reducepatch(...,'verbose')
nfv = reducepatch(f,v,r)
[nf,nv] = reducepatch(...)
```

Description

`reducepatch(p,r)` reduces the number of faces of the patch identified by handle `p`, while attempting to preserve the overall shape of the original object. The MATLAB software interprets the reduction factor `r` in one of two ways depending on its value:

- If `r` is less than 1, `r` is interpreted as a fraction of the original number of faces. For example, if you specify `r` as 0.2, then the number of faces is reduced to 20% of the number in the original patch.
- If `r` is greater than or equal to 1, then `r` is the target number of faces. For example, if you specify `r` as 400, then the number of faces is reduced until there are 400 faces remaining.

`nfv = reducepatch(p,r)` returns the reduced set of faces and vertices but does not set the `Faces` and `Vertices` properties of patch `p`. The struct `nfv` contains the faces and vertices after reduction.

`nfv = reducepatch(fv,r)` performs the reduction on the faces and vertices in the struct `fv`.

`nfv = reducepatch(p) or nfv = reducepatch(fv)` uses a reduction value of `0.5`.

`reducepatch(...,'fast')` assumes the vertices are unique and does not compute shared vertices.

`reducepatch(...,'verbose')` prints progress messages to the command window as the computation progresses.

`nfv = reducepatch(f,v,r)` performs the reduction on the faces in `f` and the vertices in `v`.

`[nf,nv] = reducepatch(...)` returns the faces and vertices in the arrays `nf` and `nv`.

Tips

If the patch contains nonshared vertices, MATLAB computes shared vertices before reducing the number of faces. If the faces of the patch are not triangles, MATLAB triangulates the faces before reduction. The faces returned are always defined as triangles.

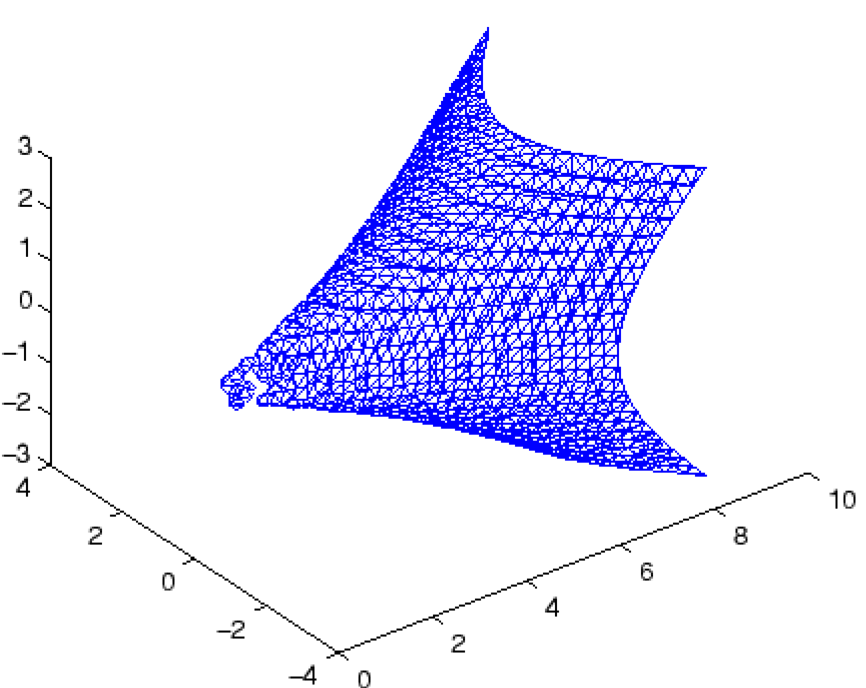
The number of output triangles may not be exactly the number specified with the reduction factor argument (`r`), particularly if the faces of the original patch are not triangles.

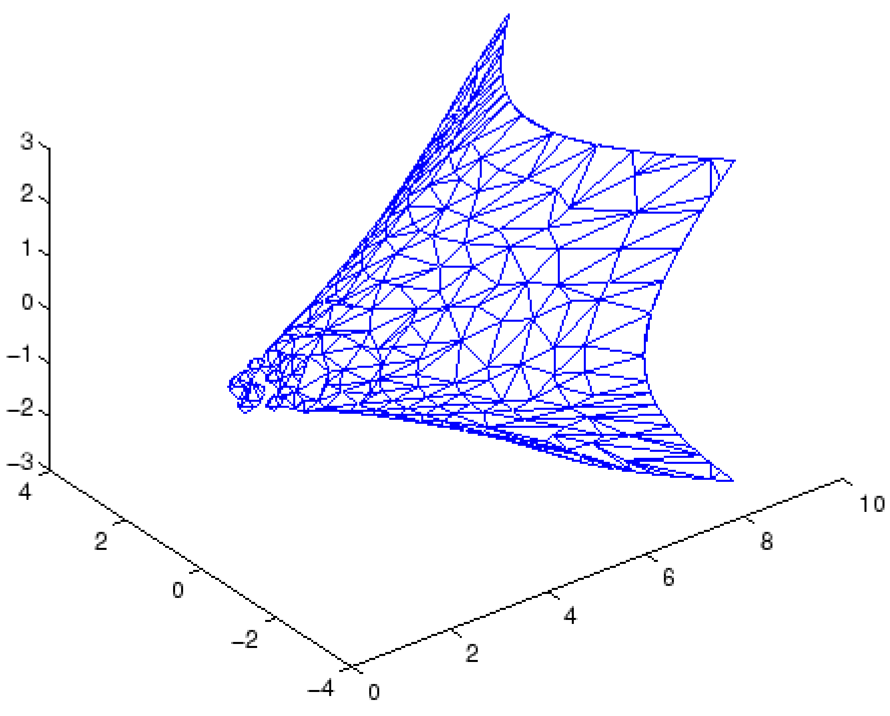
Examples

This example illustrates the effect of reducing the number of faces to only 15% of the original value.

```
[x,y,z,v] = flow;
p = patch(isosurface(x,y,z,v,-3));
set(p,'facecolor','w','EdgeColor','b');
daspect([1,1,1])
view(3)
figure;
h = axes;
p2 = copyobj(p,h);
reducepatch(p2,0.15)
daspect([1,1,1])
view(3)
```

Before Reduction


3
2
1
0
−1
−2
−3
4
2
0
−2
−4
0
2
4
6
8
10



See Also

`isosurface` | `isocaps` | `isonormals` | `smooth3` | `subvolume` | `reducevolume`

How To

- Vector Field Displayed with Cone Plots

Purpose

Reduce number of elements in volume data set

Syntax

```
[nx,ny,nz,nv] = reducevolume(X,Y,Z,V,[Rx,Ry,Rz])
[nx,ny,nz,nv] = reducevolume(V,[Rx,Ry,Rz])
nv = reducevolume(...)
```

Description

`[nx,ny,nz,nv] = reducevolume(X,Y,Z,V,[Rx,Ry,Rz])` reduces the number of elements in the volume by retaining every `Rx`th element in the x direction, every `Ry`th element in the y direction, and every `Rz`th element in the z direction. If a scalar `R` is used to indicate the amount or reduction instead of a three-element vector, the MATLAB software assumes the reduction to be `[R R R]`.

The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The reduced volume is returned in `nv`, and the coordinates of the reduced volume are returned in `nx`, `ny`, and `nz`.

`[nx,ny,nz,nv] = reducevolume(V,[Rx,Ry,Rz])` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:n,1:m,1:p)`, where `[m,n,p] = size(V)`.

`nv = reducevolume(...)` returns only the reduced volume.

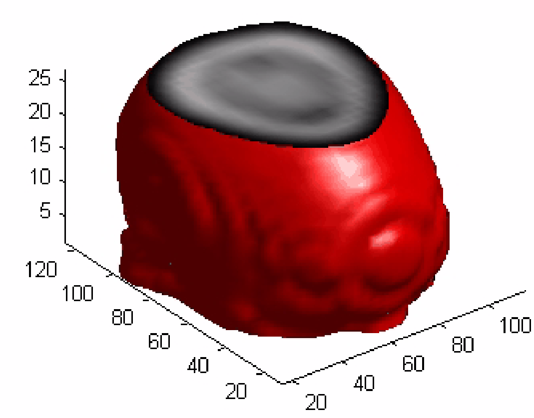
Examples

This example uses a data set that is a collection of MRI slices of a human skull. This data is processed in a variety of ways:

- The 4-D array is squeezed (`squeeze`) into three dimensions and then reduced (`reducevolume`) so that what remains is every fourth element in the x and y directions and every element in the z direction.
- The reduced data is smoothed (`smooth3`).
- The outline of the skull is an isosurface generated as a patch (`p1`) whose vertex normals are recalculated to improve the appearance when lighting is applied (`patch`, `isosurface`, `isonormals`).
- A second patch (`p2`) with an interpolated face color draws the end caps (`FaceColor`) `isocaps`).
- The view of the object is set (`view`, `axis`, `daspect`).

- A 100-element grayscale colormap provides coloring for the end caps (colormap).
- Adding a light to the right of the camera illuminates the object (camlight, lighting).

```
load mri
D = squeeze(D);
[x,y,z,D] = reducevolume(D,[4,4,1]);
D = smooth3(D);
p1 = patch(isosurface(x,y,z,D, 5,'verbose'),...
    'FaceColor','red','EdgeColor','none');
isonormals(x,y,z,D,p1);
p2 = patch(isocaps(x,y,z,D, 5),...
    'FaceColor','interp','EdgeColor','none');
view(3); axis tight; daspect([1,1,.4])
colormap(gray(100))
camlight; lighting gouraud
```



See Also

isosurface | isocaps | isonormals | smooth3 | subvolume | reducepatch

refresh

Purpose Redraw current figure

Syntax

```
refresh
refresh(h)
```

Description

`refresh` erases and redraws the current figure.

`refresh(h)` redraws the figure identified by `h`.

refreshdata

Purpose

Refresh data in graph when data source is specified

Syntax

```
refreshdata
refreshdata(figure_handle)
refreshdata(object_handles)
refreshdata(object_handles,'workspace')
```

Description

`refreshdata` evaluates any data source properties (`XDataSource`, `YDataSource`, or `ZDataSource`) on all objects in graphs in the current figure. If the specified data source has changed, the MATLAB software updates the graph to reflect this change.

Note The variable assigned to the data source property must be in the base workspace or you must specify the *`workspace`* option as `'caller'`.

`refreshdata(figure_handle)` refreshes the data of the objects in the specified figure.

`refreshdata(object_handles)` refreshes the data of the objects specified in `object_handles` or the children of those objects. Therefore, `object_handles` can contain figure, axes, or plot object handles.

`refreshdata(object_handles,'`*`workspace`*`')` enables you to specify whether the data source properties are evaluated in the base workspace or the workspace of the function in which `refreshdata` was called. *`workspace`* is a string that can be

- `base` — Evaluate the data source properties in the base workspace.
- `caller` — Evaluate the data source properties in the workspace of the function that called refreshdata.

Tips

The Linked Plots feature (see documentation for `linkdata`) sets up data sources for graphs and synchronizes them with the workspace variables they display. When you use this feature, you do not also need to call

`refreshdata`, as it is essentially automatically triggered every time a data source changes.

If you are not using the Linked Plots feature, you need to set the `XDataSource`, `YDataSource`, and/or `ZDataSource` properties of a graph in order to use `refreshdata`. You can do that programmatically, as shown in the examples below, or use the Property Editor, one of the plotting tools. In the Property Editor, select the graph (e.g., a lineseries object) and type in (or select from the drop-down choices) the name(s) of the workspace variable(s) from which you want the plot to refresh, in the fields labelled **X Data Source**, **Y Data Source**, and/or **Z Data Source**. The call to `refreshdata` causes the graph to update.

Examples

Plot a sine wave, identify data sources, and then modify its `YDataSource`:

```
x = 0:.1:8;
y = sin(x);
h = plot(x,y)
set(h,'YDataSource','y')
set(h,'XDataSource','x')
y = sin(x.^3);
refreshdata
```

Create a surface plot, identify a `ZDataSource` for it, and change the data to a different size.

```
Z = peaks(5);
h = surf(Z)
set(h,'ZDataSource','Z')
pause(3)
Z = peaks(25);
refreshdata
```

How To

- plot objects
- "Updating Plot Object Axis and Color Data"

Purpose

Match regular expression (case sensitive)

Syntax

```
startIndex = regexp(str,expression)
[startIndex,endIndex] = regexp(str,expression)

out = regexp(str,expression,outkey)
[out1,...,outN] =
regexp(str,expression,outkey1,...,outkeyN)

___= regexp(___,option1,...,optionM)
```

Description

`startIndex = regexp(str,expression)` returns the starting index of each substring of `str` that matches the character patterns specified by the regular expression. If there are no matches, `startIndex` is an empty array.

`[startIndex,endIndex] = regexp(str,expression)` returns the starting and ending indices of all matches.

`out = regexp(str,expression,outkey)` returns the output specified by `outkey`. For example, if `outkey` is `'match'`, then `regexp` returns the substrings that match the expression rather than their starting indices.

`[out1,...,outN] = regexp(str,expression,outkey1,...,outkeyN)` returns the outputs specified by multiple output keywords, in the specified order. For example, if you specify `'match'`,`'tokens'`, then `regexp` returns substrings that match the entire expression and tokens that match parts of the expression.

`___= regexp(___,option1,...,optionM)` modifies the search using the specified option flags. For example, specify `'ignorecase'` to perform a case-insensitive match. You can include any of the inputs and request any of the outputs from previous syntaxes.

Input Arguments

str - Input text

string | cell array of strings

Input text, specified as a string or a cell array of strings. Each string can be of any length and contain any characters.

If `str` and `expression` are both cell arrays, they must have the same dimensions.

Data Types

`char` | `cell`

expression - Regular expression

string | cell array of strings

Regular expression, specified as a string or a cell array of strings. Each expression can contain characters, metacharacters, operators, tokens, and flags that specify patterns to match in `str`.

The following tables describe the elements of regular expressions.

Metacharacters

Metacharacters represent letters, letter ranges, digits, and space characters. Use them to construct a generalized pattern of characters.

Metacharacter	Description	Example
`.`	Any single character, including white space	`'..ain'` matches sequences of five consecutive characters that end with `'ain'`.
[$c_1c_2c_3$]	Any character contained within the brackets	`'[rp]ain'` matches `'rain'` or `'pain'`.
[^$c_1c_2c_3$]	Any character not contained within the brackets	`'[^rp]ain'` matches all four-letter sequences that end in `'ain'`, except `'rain'` and `'pain'`. For example, it matches `'gain'`, `'lain'`, or `'vain'`.

Metacharacter	Description	Example
[c_1-c_2]	Any character in the range of c_1 through c_2	'[A-G]' matches a single character in the range of A through G.
\w	Any alphabetic, numeric, or underscore character. For English character sets, \w is equivalent to [a-zA-Z_0-9]	'\w*' identifies a word.
\W	Any character that is not alphabetic, numeric, or underscore. For English character sets, \W is equivalent to [^a-zA-Z_0-9]	'\W*' identifies a substring that is not a word.
\s	Any white-space character; equivalent to [\f\n\r\t\v]	'\w*n\s' matches words that end with the letter n, followed by a white-space character.
\S	Any non-white-space character; equivalent to [^ \f\n\r\t\v]	'\d\S' matches a numeric digit followed by any non-white-space character.
\d	Any numeric digit; equivalent to [0-9]	'\d*' matches any number of consecutive digits.
\D	Any nondigit character; equivalent to [^0-9]	'\w*\D\>' matches words that do not end with a numeric digit.
\oN or \o{N}	Character of octal value N	'\o{40}' matches the space character, defined by octal 40.
\xN or \x{N}	Character of hexadecimal value N	'\x2C' matches the comma character, defined by hex 2C.

Character Representation

Operator	Description
\a	Alarm (beep)
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
char	Any character with special meaning in regular expressions that you want to match literally (for example, use \\ to match a single backslash)

Quantifiers

Quantifiers specify the number of times a string pattern must occur in the matching string.

Quantifier	Matches the expression when it occurs...	Example
expr*	0 or more times consecutively.	'\w*' matches a word of any length.
expr?	0 times or 1 time.	'\w*(\.m)?' matches words that optionally end with the extension .m.
expr+	1 or more times consecutively.	'<img src="\w+\.gif">' matches an <img> HTML tag when the file name contains one or more characters.
expr{m,n}	At least m times, but no more than n times consecutively. {0,1} is equivalent to ?.	'\S{4,8}' matches between four and eight non-white-space characters.

Quantifier	Matches the expression when it occurs...	Example
expr{m,}	At least m times consecutively. {0,} and {1,} are equivalent to * and +, respectively.	'<a href="\w{1,}\.html">' matches an <a> HTML tag when the file name contains one or more characters.
expr{n}	Exactly n times consecutively. Equivalent to {n,n}.	'\d{4}' matches four consecutive digits.

Quantifiers can appear in three modes, described in the following table. *q* represents any of the quantifiers in the previous table.

Mode	Description	Example
expr*q*	Greedy expression: match as many characters as possible.	Given the string '<tr><td> text </td>', the expression '</?t.*>' matches all characters between <tr and /td>: '<tr><td> text </td>'
expr*q*?	Lazy expression: match as few characters as necessary.	Given the string '<tr><td> text </td>', the expression '</?t.*?>' ends each match at the first occurrence of the closing bracket (>): '<tr>' '<td>' '</td>'
expr*q*+	Possessive expression: match as much as possible, but do not rescan any portions of the string.	Given the string '<tr><td> text </td>', the expression '</?t.*+>' does not return any matches, because the closing bracket is captured using .*, and is not rescanned.

Grouping Operators

Grouping operators allow you to capture tokens, apply one operator to multiple elements, or disable backtracking in a specific group.

<table>
<tr><th>Grouping Operator</th><th>Description</th><th>Example</th></tr>
<tr><td>(expr)</td><td>Group elements of the expression and capture tokens.</td><td>'Joh?n\s(\w*)' captures a token that contains the last name of any person with the first name John or Jon.</td></tr>
<tr><td>(?:expr)</td><td>Group, but do not capture tokens.</td><td>'(?:[aeiou][^aeiou]){2}' matches two consecutive patterns of a vowel followed by a nonvowel, such as 'anon'.

Without grouping, '[aeiou][^aeiou]{2}'matches a vowel followed by two nonvowels.</td></tr>
<tr><td>(?>expr)</td><td>Group atomically. Do not backtrack within the group to complete the match, and do not capture tokens.</td><td>'A(?>.*)Z' does not match 'AtoZ', although 'A(?:.*)Z' does. Using the atomic group, Z is captured using .* and is not rescanned.</td></tr>
<tr><td>(expr1|expr2)</td><td>Match expression expr1 or expression expr2.

If there is a match with expr1, then expr2 is ignored.

You can include ?: or ?> after the opening parenthesis to suppress tokens or group atomically.</td><td>'(let|tel)\w+' matches words in a string that start with let or tel.</td></tr>
</table>

Anchors

Anchors in the expression match the beginning or end of the string or word.

Anchor	Matches the...	Example
`^expr`	Beginning of the input string.	`'^M\w*'` matches a word starting with `M` at the beginning of the string.
`expr$`	End of the input string.	`'\w*m$'` matches words ending with `m` at the end of the string.
`\<expr`	Beginning of a word.	`'\<n\w*'` matches any words starting with `n`.
`expr\>`	End of a word.	`'\w*e\>'` matches any words ending with `e`.

Lookaround Assertions

Lookaround assertions look for string patterns that immediately precede or follow the intended match, but are not part of the match.

The pointer remains at the current location, and characters that correspond to the `test` expression are not captured or discarded. Therefore, lookahead assertions can match overlapping character groups.

Lookaround Assertion	Description	Example
`expr(?=test)`	Look ahead for characters that match `test`.	`'\w*(?=ing)'` matches strings that are followed by `ing`, such as `'Fly'` and `'fall'` in the input string `'Flying, not falling.'`
`expr(?!test)`	Look ahead for characters that do not match `test`.	`'i(?!ng)'` matches instances of the letter `i` that are not followed by `ng`.

Lookaround Assertion	Description	Example
`(?<=test)expr`	Look behind for characters that match `test`..	`'(?<=re)\w*'` matches strings that follow `'re'`, such as `'new'`, `'use'`, and `'cycle'` in the input string `'renew, reuse, recycle'`
`(?<!test)expr`	Look behind for characters that do not match `test`.	`'(?<!\d)(\d)(?!\d)'` matches single-digit numbers (digits that do not precede or follow other digits).

If you specify a lookahead assertion *before* an expression, the operation is equivalent to a logical `AND`.

Operation	Description	Example
`(?=test)expr`	Match both `test` and `expr`.	`'(?=[a-z])[^aeiou]'` matches consonants.
`(?!test)expr`	Match `expr` and do not match `test`.	`'(?![aeiou])[a-z]'` matches consonants.

Logical and Conditional Operators

Logical and conditional operators allow you to test the state of a given condition, and then use the outcome to determine which string, if any, to match next. These operators support logical `OR`, and `if` or `if/else` conditions.

Conditions can be tokens, lookaround operators, or dynamic expressions of the form `(?@cmd)`. Dynamic expressions must return a logical or numeric value.

<table>
<tr><th>Conditional Operator</th><th>Description</th><th>Example</th></tr>
<tr><td>expr1|expr2</td><td>Match expression expr1 or expression expr2.

If there is a match with expr1, then expr2 is ignored.</td><td>'(let|tel)\w+' matches words in a string that start with let or tel.</td></tr>
<tr><td>(?(cond)expr)</td><td>If condition cond is true, then match expr.</td><td>'(?(?@ispc)[A-Z]:\\)' matches a drive name, such as C:\, when run on a Windows system.</td></tr>
<tr><td>(?(cond)expr1|expr2)</td><td>If condition cond is true, then match expr1. Otherwise, match expr2.</td><td>'Mr(s?)\..*?(?(1)her|his) \w*' matches strings that include her when the string begins with Mrs, or that include his when the string begins with Mr.</td></tr>
</table>

Token Operators

Tokens are portions of the matched text that you define by enclosing part of the regular expression in parentheses. You can refer to a token by its sequence in the string (an ordinal token), or assign names to tokens for easier code maintenance and readable output.

Ordinal Token Operator	Description	Example
(expr)	Capture in a token the characters that match the enclosed expression.	'Joh?n\s(\w*)' captures a token that contains the last name of any person with the first name John or Jon.
\N	Match the Nth token.	'<(\w+).*>.*</\1>' captures tokens for HTML tags, such

<table>
<tr><th>Ordinal Token Operator</th><th>Description</th><th>Example</th></tr>
<tr><td></td><td></td><td>as 'title' from the string '<title>Some text</title>'.</td></tr>
<tr><td>(?(N)expr1|expr2)</td><td>If the Nth token is found, then match expr1. Otherwise, match expr2.</td><td>'Mr(s?)\..*?(?(1)her|his) \w*' matches strings that include her when the string begins with Mrs, or that include his when the string begins with Mr.</td></tr>
</table>

<table>
<tr><th>Named Token Operator</th><th>Description</th><th>Example</th></tr>
<tr><td>(?<name>expr)</td><td>Capture in a named token the characters that match the enclosed expression.</td><td>'(?<month>\d+)-(?<day>\d+)-(?<yr>\d+)' creates named tokens for the month, day, and year in an input date string of the form mm-dd-yy.</td></tr>
<tr><td>\k<name></td><td>Match the token referred to by name.</td><td>'<(?<tag>\w+).*>.*</\k<tag>>' captures tokens for HTML tags, such as 'title' from the string '<title>Some text</title>'.</td></tr>
<tr><td>(?(name)expr1|expr2)</td><td>If the named token is found, then match expr1. Otherwise, match expr2.</td><td>'Mr(?<sex>s?)\..*?(?(sex)her|his) \w*' matches strings that include her when the string begins with Mrs, or that include his when the string begins with Mr.</td></tr>
</table>

Note If an expression has nested parentheses, MATLAB captures tokens that correspond to the outermost set of parentheses. For example, given the search pattern `'(and(y|rew))'`, MATLAB creates a token for `'andrew'` but not for `'y'` or `'rew'`.

Dynamic Regular Expressions

Dynamic expressions allow you to execute a MATLAB command or a regular expression to determine the text to match.

The parentheses that enclose dynamic expressions do *not* create a capturing group.

Operator	Description	Example
`(??expr)`	Parse `expr` and include the resulting string in the match expression. When parsed, `expr` must correspond to a complete, valid regular expression. Dynamic expressions that use the backslash escape character (\) require two backslashes: one for the initial parsing of `expr`, and one for the complete match.	`'^(\d+)((??\\w{$1}))'` determines how many characters to match by reading a digit at the beginning of the string. The dynamic expression is enclosed in a second set of parentheses so that the resulting match is captured in a token. For instance, matching `'5XXXXX'` captures tokens for `'5'` and `'XXXXX'`.
`(??@cmd)`	Execute the MATLAB command represented by `cmd`, and include the string returned by the command in the match expression.	`'(.{2,}).?(??@fliplr($1))'` finds palindromes that are at least four characters long, such as `'abba'`.
`(?@cmd)`	Execute the MATLAB command represented by `cmd`, but discard any output the command returns. (Helpful for diagnosing regular expressions.)	`'\w*?(\w)(?@disp($1))\1\w*'` matches words that include double letters (such as `pp`), and displays intermediate results.

Within dynamic expressions, use the following operators to define replacement strings.

Replacement String Operator	Description
$& or $0	Portion of the input string that is currently a match
$`	Portion of the input string that precedes the current match
$'	Portion of the input string that follows the current match (use $'' to represent the string $')
$N	Nth token
$<name>	Named token
${cmd}	String returned when MATLAB executes the command, cmd

Comments

Characters	Description	Example
(?#comment)	Insert a comment in the regular expression. The comment text is ignored when matching the input string.	'(?# Initial digit)\<\d\w+' includes a comment, and matches words that begin with a number.

Search Flags

Search flags modify the behavior for matching expressions. An alternative to using a search flag within an expression is to pass an option input argument.

Flag	Description
(?-i)	Match letter case (default for regexp and regexprep).
(?i)	Do not match letter case (default for regexpi).
(?s)	Match dot (.) in the pattern string with any character (default).
(?-s)	Match dot in the pattern with any character that is not a newline character.
(?-m)	Match the ^ and $ metacharacters at the beginning and end of a string (default).
(?m)	Match the ^ and $ metacharacters at the beginning and end of a line.
(?-x)	Include space characters and comments when matching (default).
(?x)	Ignore space characters and comments when matching. Use '\ ' and '\#' to match space and # characters.

The expression that the flag modifies can appear either after the parentheses, such as

```
(?i)\w*
```

or inside the parentheses and separated from the flag with a colon (:), such as

```
(?i:\w*)
```

The latter syntax allows you to change the behavior for part of a larger expression.

Data Types
char | cell

outkey - Keyword that indicates which outputs to return

'start' | 'end' | 'tokenExtents' | 'match' | 'tokens' | 'names' | 'split'

Keyword that indicates which outputs to return, specified as one of the following strings.

Output Keyword	Returns
'start'	Starting indices of all matches, startIndex
'end'	Ending indices of all matches, endIndex
'tokenExtents'	Starting and ending indices of all tokens
'match'	Text of each substring that matches the pattern in expression
'tokens'	Text of each captured token in str
'names'	Name and text of each named token
'split'	Text of nonmatching substrings of str

Data Types
char

option - Search option

'once' | 'warnings' | 'ignorecase' | 'emptymatch' | 'dotexceptnewline' | 'lineanchors'

Search option, specified as a string. Options come in pairs: one option that corresponds to the default behavior, and one option that allows you to override the default. Specify only one option from a pair. Options can appear in any order.

Default	Override	Description
'all'	'once'	Match the expression as many times as possible (default), or only once.
'nowarnings'	'warnings'	Suppress warnings (default), or display them.

Default	Override	Description
'matchcase'	'ignorecase'	Match letter case (default), or ignore case.
'noemptymatch'	'emptymatch'	Ignore zero length matches (default), or include them.
'dotall'	'dotexceptnewline'	Match dot with any character (default), or all except newline (`\n`).
'stringanchors'	'lineanchors'	Apply `^` and `$` metacharacters to the beginning and end of a string (default), or to the beginning and end of a line.
'literalspacing'	'freespacing'	Include space characters and comments when matching (default), or ignore them. With `freespacing`, use `'\ '` and `'\#'` to match space and # characters.

Data Types
`char`

Output Arguments

startIndex - Starting index of each match
row vector | cell array of row vectors

Starting indices of each match, returned as a row vector or cell array, as follows:

- If `str` and `expression` are both strings, the output is a row vector (or, if there are no matches, an empty array).
- If `str` or `expression` is a cell array of strings, the output is a cell array of row vectors. The output cell array has the same dimensions as the input cell array.
- If `str` and `expression` are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

endIndex - Ending index of each match
row vector | cell array of row vectors

Ending index of each match, returned as a row vector or cell array, as follows:

- If `str` and `expression` are both strings, the output is a row vector (or, if there are no matches, an empty array).
- If `str` or `expression` is a cell array of strings, the output is a cell array of row vectors. The output cell array has the same dimensions as the input cell array.
- If `str` and `expression` are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

out - Information about matches

numeric array | cell array | structure array

Information about matches, returned as a numeric, cell, or structure array. The information in the output depends upon the value you specify for `outkey`, as follows.

<table>
<tr><th>Output Keyword</th><th>Output Description</th><th>Output Type and Dimensions</th></tr>
<tr><td>'start'</td><td>Starting indices of matches</td><td rowspan="2">For both 'start' and 'end':
• If str and expression are both strings, the output is a row vector (or, if there are no matches, an empty array).
• If str or expression is a cell array of strings, the output is a cell array of row vectors. The output cell array has the same dimensions as the input cell array.
• If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.</td></tr>
<tr><td>'end'</td><td>Ending indices of matches</td></tr>
</table>

Output Keyword	Output Description	Output Type and Dimensions
'tokenExtents'	Starting and ending indices of all tokens	By default, when returning all matches: • If str and expression are both strings, the output is a 1-by-n-cell array, where n is the number of matches. Each cell contains an m-by-2 numeric array of indices, where m is the number of tokens in the match. • If str or expression is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-n cell array, where each inner cell contains an m-by-2 numeric array. • If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions. When you specify the 'once' option to return only one match, the output is either an m-by-2 numeric array or a cell array with the same dimensions as str and/or expression. If a token is expected at a particular index N, but is not found, then MATLAB returns extents for that token of [N,N-1].

Output Keyword	Output Description	Output Type and Dimensions
`'match'`	Text of each substring that matches the pattern in `expression`	By default, when returning all matches: • If `str` and `expression` are both strings, the output is a 1-by-`n`-cell array of strings, where `n` is the number of matches. • If `str` or `expression` is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-`n` cell array of strings. • If `str` and `expression` are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions. When you specify the `'once'` option to return only one match, the output is either a string or a cell array of strings with the same dimensions as `str` and `expression`.
`'tokens'`	Text of each captured token in `str`	By default, when returning all matches: • If `str` and `expression` are both strings, the output is a 1-by-`n`-cell array, where `n` is the number of matches. Each cell contains a 1-by-`m` cell array of strings, where `m` is the number of tokens in the match. • If `str` or `expression` is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-`n` cell array, where each inner cell contains a 1-by-`m` cell array of strings.

Output Keyword	Output Description	Output Type and Dimensions
		• If `str` and `expression` are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions. When you specify the `'once'` option to return only one match, the output is a 1-by-`m` cell array of strings or a cell array that has the same dimensions as `str` and/or `expression`. If a token is expected at a particular index, but is not found, then MATLAB returns an empty string for the token, `''`.
`'names'`	Name and text of each named token	For all matches: • If `str` and `expression` are both strings, the output is a 1-by-`n` structure array, where `n` is the number of matches. The structure field names correspond to the token names. • If `str` or `expression` is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-`n` structure array. • If `str` and `expression` are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.
`'split'`	Text of nonmatching substrings of `str`	For all matches: • If `str` and `expression` are both strings, the output is a 1-by-`n`-cell array of strings, where `n` is the number of nonmatching strings.

Output Keyword	Output Description	Output Type and Dimensions
		• If `str` or `expression` is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-n cell array of strings. • If `str` and `expression` are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

Examples

Find Patterns in Single Strings

Find words that start with `c`, end with `t`, and contain one or more vowels between them.

```
str = 'bat cat can car coat court CUT ct CAT-scan';
expression = 'c[aeiou]+t';
startIndex = regexp(str,expression)
```

```
startIndex =
     5    17
```

The regular expression `'c[aeiou]+t'` specifies this pattern:

- `c` must be the first character.
- `c` must be followed by one of the characters inside the brackets, `[aeiou]`.
- The bracketed pattern must occur one or more times, as indicated by the `+` operator.
- `t` must be the last character, with no characters between the bracketed pattern and the `t`.

Values in `startIndex` indicate the index of the first character of each word that matches the regular expression. The matching word `cat`

starts at index 5, and `coat` starts at index 17. The words `CUT` and `CAT` do not match because they are uppercase.

Find Patterns in Multiple Strings

Find the location of capital letters and spaces within strings in a cell array.

```
str = {'Madrid, Spain','Romeo and Juliet','MATLAB is great'};
capExpr = '[A-Z]';
spaceExpr = '\s';

capStartIndex = regexp(str,capExpr);
spaceStartIndex = regexp(str,spaceExpr);
```

`capStartIndex` and `spaceStartIndex` are cell arrays because the input `str` is a cell array.

View the indices for the capital letters.

```
celldisp(capStartIndex)
```

```
capStartIndex{1} =
     1     9

capStartIndex{2} =
     1    11

capStartIndex{3} =
     1     2     3     4     5     6
```

View the indices for the spaces.

```
celldisp(spaceStartIndex)
```

```
spaceStartIndex{1} =
     8

spaceStartIndex{2} =
```

```
     6    10

spaceStartIndex{3} =
     7    10
```

Return Substrings Using 'match' Keyword

Capture words within a string that contain the letter `x`.

```
str = 'EXTRA! The regexp function helps you relax.';
expression = '\w*x\w*';
matchStr = regexp(str,expression,'match')
```

```
matchStr =
    'regexp'    'relax'
```

The regular expression `'\w*x\w*'` specifies that the string:

- Begins with any number of alphanumeric or underscore characters, `\w*`.
- Contains the lowercase letter `x`.
- Ends with any number of alphanumeric or underscore characters after the `x`, including none, as indicated by `\w*`.

Split String at Delimiter Using 'split' Keyword

Split a string into several substrings, where each substring is delimited by a `^` character.

```
str = ['Split ^this string into ^several pieces'];
expression = '\^';
splitStr = regexp(str,expression,'split')
```

```
splitStr =
    'Split '    'this string into '    'several pieces'
```

Because the carat symbol has special meaning in regular expressions, precede it with the escape character, a backslash (`\`). To split a string

at other delimiters, such as a semicolon, you do not need to include the backslash.

Return Both Matching and Nonmatching Substrings

Capture parts of a string that match a regular expression using the `'match'` keyword, and the remaining parts that do not match using the `'split'` keyword.

```
str = 'She sells sea shells by the seashore.';
expression = '[Ss]h.';
[match,noMatch] = regexp(str,expression,'match','split')
```

```
match = 
    'She'    'she'    'sho'

noMatch = 
    ''    ' sells sea '    'lls by the sea'    're.'
```

The regular expression `'[Ss]h.'` specifies that:

- `S` or `s` is the first character.
- `h` is the second character.
- The third character can be anything, including a space, as indicated by the dot (`.`).

When the first (or last) character in a string matches a regular expression, the first (or last) return value from the `'split'` keyword is an empty string.

Optionally, reassemble the original string from the substrings.

```
combinedStr = strjoin(noMatch,match)
```

```
combinedStr =
She sells sea shells by the seashore.
```

Capture Substrings of Matches Using Ordinal Tokens

Find the names of HTML tags by defining a token within a regular expression. Tokens are indicated with parentheses, ().

```
str = '<title>My Title</title><p>Here is some text.</p>';
expression = '<(\w+).*>.*</\1>';
[tokens,matches] = regexp(str,expression,'tokens','match');
```

The regular expression <(\w+).*>.*</\1> specifies this pattern:

- <(\w+) finds an opening angle bracket followed by one or more alphanumeric or underscore characters. Enclosing \w+ in parentheses captures the name of the HTML tag in a token.
- .*> finds any number of additional characters, such as HTML attributes, and a closing angle bracket.
- </\1> finds the end tag corresponding to the first token (indicated by \1). The end tag has the form <*tagname*>.

View the tokens and matching substrings.

```
celldisp(tokens)
```

```
tokens{1}{1} =
   title

tokens{2}{1} =
   p
```

```
celldisp(matches)
```

```
matches{1} =
   <title>My Title</title>

matches{2} =
   <p>Here is some text.</p>
```

Capture Substrings of Matches Using Named Tokens

Parse dates that can appear with either the day or the month first, in these forms: `mm/dd/yyyy` or `dd-mm-yyyy`. Use named tokens to identify each part of the date.

```
str = '01/11/2000  20-02-2020  03/30/2000  16-04-2020';
expression = ['(?<month>\d+)/(?<day>\d+)/(?<year>\d+)|'...
              '(?<day>\d+)-(?<month>\d+)-(?<year>\d+)'];
tokenNames = regexp(str,expression,'names');
```

The regular expression specifies this pattern:

- `(?<name>\d+)` finds one or more numeric digits and assigns the result to the token indicated by *`name`*.
- `|` is the logical `or` operator, which indicates that there are two possible patterns for dates. In the first pattern, slashes (`/`) separate the tokens. In the second pattern, hyphens (`-`) separate the tokens.

View the named tokens.

```
for k = 1:length(tokenNames)
   disp(tokenNames(k))
end
```

```
    month: '01'
      day: '11'
     year: '2000'

    month: '02'
      day: '20'
     year: '2020'

    month: '03'
      day: '30'
     year: '2000'

    month: '04'
```

```
      day: '16'
     year: '2020'
```

Perform Case-Insensitive Matches

Find both uppercase and lowercase instances of a word.

By default, `regexp` performs case-sensitive matching.

```
str = 'A string with UPPERCASE and lowercase text.';
expression = '\w*case';
matchStr = regexp(str,expression,'match')
```

```
matchStr =
    'lowercase'
```

The regular expression specifies that the string:

- Begins with any number of alphanumeric or underscore characters, `\w*`.
- Ends with the literal text `case`.

The `regexpi` function uses the same syntax as `regexp`, but performs case-insensitive matching.

```
matchWithRegexpi = regexpi(str,expression,'match')
```

```
matchWithRegexpi =
    'UPPERCASE'    'lowercase'
```

Alternatively, disable case-sensitive matching for `regexp` using the `'ignorecase'` option.

```
matchWithIgnorecase = regexp(str,expression,'match','ignorecase')
```

```
matchWithIgnorecase =
    'UPPERCASE'    'lowercase'
```

For multiple expressions, disable case-sensitive matching for selected expressions using the `(?i)` search flag.

```
expression = {'(?-i)\w*case';...
              '(?i)\w*case'};
matchStr = regexp(str,expression,'match');
celldisp(matchStr)
```

```
matchStr{1}{1} =
   lowercase

matchStr{2}{1} =
   UPPERCASE

matchStr{2}{2} =
   lowercase
```

Parse Strings with Newline Characters

Create a string that contains a newline, `\n`, and parse the string using a regular expression.

```
str = sprintf('abc\n de');
expression = '.*';
matchStr = regexp(str,expression,'match')
```

```
matchStr =
   [1x7 char]
```

By default, the dot (`.`) matches every character, including the newline, and returns a single match that is equivalent to the original string.

Exclude newline characters from the match using the `'dotexceptnewline'` option. This returns separate matches for each line of text.

```
matchStrNoNewline = regexp(str,expression,'match','dotexceptnewline')
```

```
matchStrNoNewline =
    'abc'    ' de'
```

Find the first or last character of each line using the ^ or $ metacharacters and the 'lineanchors' option.

```
expression = '.$';
lastInLine = regexp(str,expression,'match','lineanchors')
```

```
lastInLine =
    'c'    'e'
```

Definitions

Tokens

Tokens are portions of the matched text that correspond to portions of the regular expression. To create tokens, enclose part of the regular expression in parentheses.

For example, this expression finds a date of the form dd-mmm-yyyy, including tokens for the day, month, and year.

```
str = 'Here is a date: 01-Apr-2020';
expression = '(\d+)-(\w+)-(\d+)';

mydate = regexp(str,expression,'tokens');
mydate{:}
```

```
ans =
    '01'    'April'    '2020'
```

You can associate names with tokens so that they are more easily identifiable:

```
str = 'Here is a date: 01-Apr-2020';
expression = '(?<day>\d+)-(?<month>\w+)-(?<year>\d+)';

mydate = regexp(str,expression,'names')
```

```
mydate =
      day: '01'
    month: 'Apr'
     year: '2020'
```

For more information, see “Tokens in Regular Expressions”.

Algorithms

MATLAB parses each input string from left to right, attempting to match the text in the string with the first element of the regular expression. During this process, MATLAB skips over any text that does not match.

When MATLAB finds the first match, it continues parsing the string to match the second piece of the expression, and so on.

Tips

- Use `strfind` to find an exact character match within a string. Use `regexp` to look for a pattern of characters.

See Also

`regexpi | regexprep | regexptranslate | strfind | strjoin | strsplit | strrep`

Concepts

- “Lookahead Assertions in Regular Expressions”
- “Dynamic Regular Expressions”

Purpose

Match regular expression (case insensitive)

Syntax

```
startIndex = regexpi(str,expression)
[startIndex,endIndex] = regexpi(str,expression)

out = regexpi(str,expression,outkey)
[out1,...,outN] =
regexpi(str,expression,outkey1,...,outkeyN)

___= regexpi(___,option1,...,optionM)
```

Description

`startIndex = regexpi(str,expression)` returns the starting index of each substring of `str` that matches the character patterns specified by the regular expression, without regard to letter case. If there are no matches, `startIndex` is an empty array.

`[startIndex,endIndex] = regexpi(str,expression)` returns the starting and ending indices of all matches.

`out = regexpi(str,expression,outkey)` returns the output specified by `outkey`. For example, if `outkey` is `'match'`, then `regexpi` returns the substrings that match the expression rather than their starting indices.

`[out1,...,outN] =`
`regexpi(str,expression,outkey1,...,outkeyN)` returns the outputs specified by multiple output keywords, in the specified order. For example, if you specify `'match'`,`'tokens'`, then `regexpi` returns substrings that match the entire expression and tokens that match parts of the expression.

`___= regexpi(___,option1,...,optionM)` modifies the search using the specified option flags. For example, specify `'matchcase'` to perform a case-sensitive match. You can include any of the inputs and request any of the outputs from previous syntaxes.

Input Arguments

str - Input text

string | cell array of strings

Input text, specified as a string or a cell array of strings. Each string can be of any length and contain any characters.

If `str` and `expression` are both cell arrays, they must have the same dimensions.

Data Types

`char | cell`

expression - Regular expression

string | cell array of strings

Regular expression, specified as a string or a cell array of strings. Each expression can contain characters, metacharacters, operators, tokens, and flags that specify patterns to match in `str`.

The following tables describe the elements of regular expressions.

Metacharacters

Metacharacters represent letters, letter ranges, digits, and space characters. Use them to construct a generalized pattern of characters.

Metacharacter	Description	Example
`.`	Any single character, including white space	`'..ain'` matches sequences of five consecutive characters that end with `'ain'`.
[$c_1c_2c_3$]	Any character contained within the brackets	`'[rp]ain'` matches `'rain'` or `'pain'`.
[^$c_1c_2c_3$]	Any character not contained within the brackets	`'[^rp]ain'` matches all four-letter sequences that end in `'ain'`, except `'rain'` and `'pain'`. For example, it matches `'gain'`, `'lain'`, or `'vain'`.

Metacharacter	Description	Example
[c_1-c_2]	Any character in the range of c_1 through c_2	'[A-G]' matches a single character in the range of A through G.
\w	Any alphabetic, numeric, or underscore character. For English character sets, \w is equivalent to [a-zA-Z_0-9]	'\w*' identifies a word.
\W	Any character that is not alphabetic, numeric, or underscore. For English character sets, \W is equivalent to [^a-zA-Z_0-9]	'\W*' identifies a substring that is not a word.
\s	Any white-space character; equivalent to [\f\n\r\t\v]	'\w*n\s' matches words that end with the letter n, followed by a white-space character.
\S	Any non-white-space character; equivalent to [^ \f\n\r\t\v]	'\d\S' matches a numeric digit followed by any non-white-space character.
\d	Any numeric digit; equivalent to [0-9]	'\d*' matches any number of consecutive digits.
\D	Any nondigit character; equivalent to [^0-9]	'\w*\D\>' matches words that do not end with a numeric digit.
\oN or \o{N}	Character of octal value N	'\o{40}' matches the space character, defined by octal 40.
\xN or \x{N}	Character of hexadecimal value N	'\x2C' matches the comma character, defined by hex 2C.

Character Representation

Operator	Description
\a	Alarm (beep)
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
char	Any character with special meaning in regular expressions that you want to match literally (for example, use \\ to match a single backslash)

Quantifiers

Quantifiers specify the number of times a string pattern must occur in the matching string.

Quantifier	Matches the expression when it occurs...	Example
expr*	0 or more times consecutively.	'\w*' matches a word of any length.
expr?	0 times or 1 time.	'\w*(\.m)?' matches words that optionally end with the extension .m.
expr+	1 or more times consecutively.	'<img src="\w+\.gif">' matches an <img> HTML tag when the file name contains one or more characters.
expr{m,n}	At least m times, but no more than n times consecutively. {0,1} is equivalent to ?.	'\S{4,8}' matches between four and eight non-white-space characters.

Quantifier	Matches the expression when it occurs...	Example
expr{m,}	At least m times consecutively. {0,} and {1,} are equivalent to * and +, respectively.	'<a href="\w{1,}\.html">' matches an <a> HTML tag when the file name contains one or more characters.
expr{n}	Exactly n times consecutively. Equivalent to {n,n}.	'\d{4}' matches four consecutive digits.

Quantifiers can appear in three modes, described in the following table. *q* represents any of the quantifiers in the previous table.

Mode	Description	Example
expr*q*	Greedy expression: match as many characters as possible.	Given the string '<tr><td> text </td>', the expression '</?t.*>' matches all characters between <tr and /td>: '<tr><td> text </td>'
expr*q*?	Lazy expression: match as few characters as necessary.	Given the string '<tr><td> text </td>', the expression '</?t.*?>' ends each match at the first occurrence of the closing bracket (>): '<tr>' '<td>' '</td>'
expr*q*+	Possessive expression: match as much as possible, but do not rescan any portions of the string.	Given the string '<tr><td> text </td>', the expression '</?t.*+>' does not return any matches, because the closing bracket is captured using .*, and is not rescanned.

Grouping Operators

Grouping operators allow you to capture tokens, apply one operator to multiple elements, or disable backtracking in a specific group.

<table>
<tr><th>Grouping Operator</th><th>Description</th><th>Example</th></tr>
<tr><td>(expr)</td><td>Group elements of the expression and capture tokens.</td><td>'Joh?n\s(\w*)' captures a token that contains the last name of any person with the first name John or Jon.</td></tr>
<tr><td>(?:expr)</td><td>Group, but do not capture tokens.</td><td>'(?:[aeiou][^aeiou]){2}' matches two consecutive patterns of a vowel followed by a nonvowel, such as 'anon'.

Without grouping, '[aeiou][^aeiou]{2}'matches a vowel followed by two nonvowels.</td></tr>
<tr><td>(?>expr)</td><td>Group atomically. Do not backtrack within the group to complete the match, and do not capture tokens.</td><td>'A(?>.*)Z' does not match 'AtoZ', although 'A(?:.*)Z' does. Using the atomic group, Z is captured using .* and is not rescanned.</td></tr>
<tr><td>(expr1|expr2)</td><td>Match expression expr1 or expression expr2.

If there is a match with expr1, then expr2 is ignored.

You can include ?: or ?> after the opening parenthesis to suppress tokens or group atomically.</td><td>'(let|tel)\w+' matches words in a string that start with let or tel.</td></tr>
</table>

Anchors

Anchors in the expression match the beginning or end of the string or word.

Anchor	Matches the...	Example
`^expr`	Beginning of the input string.	`'^M\w*'` matches a word starting with `M` at the beginning of the string.
`expr$`	End of the input string.	`'\w*m$'` matches words ending with `m` at the end of the string.
`\<expr`	Beginning of a word.	`'\<n\w*'` matches any words starting with `n`.
`expr\>`	End of a word.	`'\w*e\>'` matches any words ending with `e`.

Lookaround Assertions

Lookaround assertions look for string patterns that immediately precede or follow the intended match, but are not part of the match.

The pointer remains at the current location, and characters that correspond to the `test` expression are not captured or discarded. Therefore, lookahead assertions can match overlapping character groups.

Lookaround Assertion	Description	Example
`expr(?=test)`	Look ahead for characters that match `test`.	`'\w*(?=ing)'` matches strings that are followed by `ing`, such as `'Fly'` and `'fall'` in the input string `'Flying, not falling.'`
`expr(?!test)`	Look ahead for characters that do not match `test`.	`'i(?!ng)'` matches instances of the letter `i` that are not followed by `ng`.

Lookaround Assertion	Description	Example
`(?<=test)expr`	Look behind for characters that match `test`..	`'(?<=re)\w*'` matches strings that follow `'re'`, such as `'new'`, `'use'`, and `'cycle'` in the input string `'renew, reuse, recycle'`
`(?<!test)expr`	Look behind for characters that do not match `test`.	`'(?<!\d)(\d)(?!\d)'` matches single-digit numbers (digits that do not precede or follow other digits).

If you specify a lookahead assertion *before* an expression, the operation is equivalent to a logical `AND`.

Operation	Description	Example
`(?=test)expr`	Match both `test` and `expr`.	`'(?=[a-z])[^aeiou]'` matches consonants.
`(?!test)expr`	Match `expr` and do not match `test`.	`'(?![aeiou])[a-z]'` matches consonants.

Logical and Conditional Operators

Logical and conditional operators allow you to test the state of a given condition, and then use the outcome to determine which string, if any, to match next. These operators support logical `OR`, and `if` or `if/else` conditions.

Conditions can be tokens, lookaround operators, or dynamic expressions of the form `(?@cmd)`. Dynamic expressions must return a logical or numeric value.

<table>
<tr><th>Conditional Operator</th><th>Description</th><th>Example</th></tr>
<tr><td>expr1|expr2</td><td>Match expression expr1 or expression expr2.

If there is a match with expr1, then expr2 is ignored.</td><td>'(let|tel)\w+' matches words in a string that start with let or tel.</td></tr>
<tr><td>(?(cond)expr)</td><td>If condition cond is true, then match expr.</td><td>'(?(?@ispc)[A-Z]:\\)' matches a drive name, such as C:\, when run on a Windows system.</td></tr>
<tr><td>(?(cond)expr1|expr2)</td><td>If condition cond is true, then match expr1. Otherwise, match expr2.</td><td>'Mr(s?)\..*?(?(1)her|his) \w*' matches strings that include her when the string begins with Mrs, or that include his when the string begins with Mr.</td></tr>
</table>

Token Operators

Tokens are portions of the matched text that you define by enclosing part of the regular expression in parentheses. You can refer to a token by its sequence in the string (an ordinal token), or assign names to tokens for easier code maintenance and readable output.

Ordinal Token Operator	Description	Example
(expr)	Capture in a token the characters that match the enclosed expression.	'Joh?n\s(\w*)' captures a token that contains the last name of any person with the first name John or Jon.
\N	Match the Nth token.	'<(\w+).*>.*</\1>' captures tokens for HTML tags, such

<table>
<tr><th>Ordinal Token Operator</th><th>Description</th><th>Example</th></tr>
<tr><td></td><td></td><td>as 'title' from the string '<title>Some text</title>'.</td></tr>
<tr><td>(?(N)expr1|expr2)</td><td>If the Nth token is found, then match expr1. Otherwise, match expr2.</td><td>'Mr(s?)\..*?(?(1)her|his) \w*' matches strings that include her when the string begins with Mrs, or that include his when the string begins with Mr.</td></tr>
</table>

<table>
<tr><th>Named Token Operator</th><th>Description</th><th>Example</th></tr>
<tr><td>(?<name>expr)</td><td>Capture in a named token the characters that match the enclosed expression.</td><td>'(?<month>\d+)-(?<day>\d+)-(?<yr>\d+)' creates named tokens for the month, day, and year in an input date string of the form mm-dd-yy.</td></tr>
<tr><td>\k<name></td><td>Match the token referred to by name.</td><td>'<(?<tag>\w+).*>.*</\k<tag>>' captures tokens for HTML tags, such as 'title' from the string '<title>Some text</title>'.</td></tr>
<tr><td>(?(name)expr1|expr2)</td><td>If the named token is found, then match expr1. Otherwise, match expr2.</td><td>'Mr(?<sex>s?)\..*?(?(sex)her|his) \w*' matches strings that include her when the string begins with Mrs, or that include his when the string begins with Mr.</td></tr>
</table>

Note If an expression has nested parentheses, MATLAB captures tokens that correspond to the outermost set of parentheses. For example, given the search pattern `'(and(y|rew))'`, MATLAB creates a token for `'andrew'` but not for `'y'` or `'rew'`.

Dynamic Regular Expressions

Dynamic expressions allow you to execute a MATLAB command or a regular expression to determine the text to match.

The parentheses that enclose dynamic expressions do *not* create a capturing group.

Operator	Description	Example
`(??expr)`	Parse `expr` and include the resulting string in the match expression. When parsed, `expr` must correspond to a complete, valid regular expression. Dynamic expressions that use the backslash escape character (\) require two backslashes: one for the initial parsing of `expr`, and one for the complete match.	`'^(\d+)((??\\w{$1}))'` determines how many characters to match by reading a digit at the beginning of the string. The dynamic expression is enclosed in a second set of parentheses so that the resulting match is captured in a token. For instance, matching `'5XXXXX'` captures tokens for `'5'` and `'XXXXX'`.
`(??@cmd)`	Execute the MATLAB command represented by `cmd`, and include the string returned by the command in the match expression.	`'(.{2,}).?(??@fliplr($1))'` finds palindromes that are at least four characters long, such as `'abba'`.
`(?@cmd)`	Execute the MATLAB command represented by `cmd`, but discard any output the command returns. (Helpful for diagnosing regular expressions.)	`'\w*?(\w)(?@disp($1))\1\w*'` matches words that include double letters (such as `pp`), and displays intermediate results.

Within dynamic expressions, use the following operators to define replacement strings.

Replacement String Operator	Description
$& or $0	Portion of the input string that is currently a match
$`	Portion of the input string that precedes the current match
$'	Portion of the input string that follows the current match (use $'' to represent the string $')
$N	Nth token
$<name>	Named token
${cmd}	String returned when MATLAB executes the command, cmd

Comments

Characters	Description	Example
(?#comment)	Insert a comment in the regular expression. The comment text is ignored when matching the input string.	'(?# Initial digit)\<\d\w+' includes a comment, and matches words that begin with a number.

Search Flags

Search flags modify the behavior for matching expressions. An alternative to using a search flag within an expression is to pass an option input argument.

Flag	Description
`(?-i)`	Match letter case (default for `regexp` and `regexprep`).
`(?i)`	Do not match letter case (default for `regexpi`).
`(?s)`	Match dot (`.`) in the pattern string with any character (default).
`(?-s)`	Match dot in the pattern with any character that is not a newline character.
`(?-m)`	Match the `^` and `$` metacharacters at the beginning and end of a string (default).
`(?m)`	Match the `^` and `$` metacharacters at the beginning and end of a line.
`(?-x)`	Include space characters and comments when matching (default).
`(?x)`	Ignore space characters and comments when matching. Use `'\  '` and `'\#'` to match space and `#` characters.

The expression that the flag modifies can appear either after the parentheses, such as

```
(?i)\w*
```

or inside the parentheses and separated from the flag with a colon (`:`), such as

```
(?i:\w*)
```

The latter syntax allows you to change the behavior for part of a larger expression.

Data Types
`char | cell`

outkey - Keyword that indicates which outputs to return
'start' | 'end' | 'tokenExtents' | 'match' | 'tokens' | 'names' | 'split'

Keyword that indicates which outputs to return, specified as one of the following strings.

Output Keyword	Returns
'start'	Starting indices of all matches, startIndex
'end'	Ending indices of all matches, endIndex
'tokenExtents'	Starting and ending indices of all tokens
'match'	Text of each substring that matches the pattern in expression
'tokens'	Text of each captured token in str
'names'	Name and text of each named token
'split'	Text of nonmatching substrings of str

Data Types
char

option - Search option
'once' | 'warnings' | 'matchcase' | 'emptymatch' | 'dotexceptnewline' | 'lineanchors'

Search option, specified as a string. Options come in pairs: one option that corresponds to the default behavior, and one option that allows you to override the default. Specify only one option from a pair. Options can appear in any order.

Default	Override	Description
'all'	'once'	Match the expression as many times as possible (default), or only once.
'nowarnings'	'warnings'	Suppress warnings (default), or display them.

Default	Override	Description
'ignorecase'	'matchcase'	Ignore letter case (default), or match case.
'noemptymatch'	'emptymatch'	Ignore zero length matches (default), or include them.
'dotall'	'dotexceptnewline'	Match dot with any character (default), or all except newline (\n).
'stringanchors'	'lineanchors'	Apply ^ and $ metacharacters to the beginning and end of a string (default), or to the beginning and end of a line.
'literalspacing'	'freespacing'	Include space characters and comments when matching (default), or ignore them. With freespacing, use '\ ' and '\#' to match space and # characters.

Data Types
char

Output Arguments

startIndex - Starting index of each match
row vector | cell array of row vectors

Starting indices of each match, returned as a row vector or cell array, as follows:

- If str and expression are both strings, the output is a row vector (or, if there are no matches, an empty array).
- If str or expression is a cell array of strings, the output is a cell array of row vectors. The output cell array has the same dimensions as the input cell array.
- If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

endIndex - Ending index of each match
row vector | cell array of row vectors

Ending index of each match, returned as a row vector or cell array, as follows:

- If `str` and `expression` are both strings, the output is a row vector (or, if there are no matches, an empty array).
- If `str` or `expression` is a cell array of strings, the output is a cell array of row vectors. The output cell array has the same dimensions as the input cell array.
- If `str` and `expression` are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

out - Information about matches

numeric array | cell array | structure array

Information about matches, returned as a numeric, cell, or structure array. The information in the output depends upon the value you specify for `outkey`, as follows.

<table>
<tr><th>Output Keyword</th><th>Output Description</th><th>Output Type and Dimensions</th></tr>
<tr><td>'start'</td><td>Starting indices of matches</td><td rowspan="2">For both 'start' and 'end':
• If str and expression are both strings, the output is a row vector (or, if there are no matches, an empty array).
• If str or expression is a cell array of strings, the output is a cell array of row vectors. The output cell array has the same dimensions as the input cell array.
• If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.</td></tr>
<tr><td>'end'</td><td>Ending indices of matches</td></tr>
</table>

Output Keyword	Output Description	Output Type and Dimensions
`'tokenExtents'`	Starting and ending indices of all tokens	By default, when returning all matches: • If `str` and `expression` are both strings, the output is a 1-by-`n`-cell array, where `n` is the number of matches. Each cell contains an `m`-by-2 numeric array of indices, where `m` is the number of tokens in the match. • If `str` or `expression` is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-`n` cell array, where each inner cell contains an `m`-by-2 numeric array. • If `str` and `expression` are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions. When you specify the `'once'` option to return only one match, the output is either an `m`-by-2 numeric array or a cell array with the same dimensions as `str` and/or `expression`. If a token is expected at a particular index `N`, but is not found, then MATLAB returns extents for that token of `[N,N-1]`.

<table>
<tr><th>Output Keyword</th><th>Output Description</th><th>Output Type and Dimensions</th></tr>
<tr><td>'match'</td><td>Text of each substring that matches the pattern in expression</td><td>By default, when returning all matches:
• If str and expression are both strings, the output is a 1-by-n-cell array of strings, where n is the number of matches.
• If str or expression is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-n cell array of strings.
• If str and expression are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.
When you specify the 'once' option to return only one match, the output is either a string or a cell array of strings with the same dimensions as str and expression.</td></tr>
<tr><td>'tokens'</td><td>Text of each captured token in str</td><td>By default, when returning all matches:
• If str and expression are both strings, the output is a 1-by-n-cell array, where n is the number of matches. Each cell contains a 1-by-m cell array of strings, where m is the number of tokens in the match.
• If str or expression is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-n cell array, where each inner cell contains a 1-by-m cell array of strings.</td></tr>
</table>

Output Keyword	Output Description	Output Type and Dimensions
		• If `str` and `expression` are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions. When you specify the `'once'` option to return only one match, the output is a 1-by-`m` cell array of strings or a cell array that has the same dimensions as `str` and/or `expression`. If a token is expected at a particular index, but is not found, then MATLAB returns an empty string for the token, `''`.
`'names'`	Name and text of each named token	For all matches: • If `str` and `expression` are both strings, the output is a 1-by-`n` structure array, where `n` is the number of matches. The structure field names correspond to the token names. • If `str` or `expression` is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-`n` structure array. • If `str` and `expression` are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.
`'split'`	Text of nonmatching substrings of `str`	For all matches: • If `str` and `expression` are both strings, the output is a 1-by-`n`-cell array of strings, where `n` is the number of nonmatching strings.

Output Keyword	Output Description	Output Type and Dimensions
		• If `str` or `expression` is a cell array of strings, the output is a cell array with the same dimensions as the input cell array. Each cell contains a 1-by-`n` cell array of strings. • If `str` and `expression` are both cell arrays, they must have the same dimensions. The output is a cell array with the same dimensions.

Examples

Pattern Matching

Find words that start with `c`, end with `t`, and contain one or more vowels between them.

```
str = 'bat cat can car COAT court cut ct CAT-scan';
expression = 'c[aeiou]+t';
startIndex = regexpi(str,expression)

startIndex =
     5    17    28    35
```

Values in `startIndex` indicate the index of the first character of each word that matches the regular expression.

The regular expression `'c[aeiou]+t'` specifies this pattern:

- `c` must be the first character.
- `c` must be followed by one of the characters inside the brackets, `[aeiou]`.
- The bracketed pattern must occur one or more times, as indicated by the + operator.

- t must be the last character, with no characters between the bracketed pattern and the t.

Case-Sensitive Match

Match letter case in all or part of an expression.

By default, regexpi performs case-insensitive matching.

```
str = 'A string with UPPERCASE and lowercase text.';
expression = '\w*case';
matchStr = regexpi(str,expression,'match')

matchStr =
    'UPPERCASE'    'lowercase'
```

Use the regexp function with the same syntax as regexpi to perform case-sensitive matching.

```
matchWithRegexp = regexp(str,expression,'match')

matchWithRegexp =
    'lowercase'
```

To disable case-sensitive matching for regexp, use the 'ignorecase' option.

```
matchWithIgnorecase = regexp(str,expression,'match','ignorecase')

matchWithIgnorecase =
    'UPPERCASE'    'lowercase'
```

For multiple expressions, enable and disable case-insensitive matching for selected expressions using the (?i) and (?-i) search flags.

```
expression = {'(?-i)\w*case';...
              '(?i)\w*case'};
matchStr = regexp(str,expression,'match');
celldisp(matchStr)
```

```
matchStr{1}{1} =
   lowercase

matchStr{2}{1} =
   UPPERCASE

matchStr{2}{2} =
   lowercase
```

Definitions

Tokens

Tokens are portions of the matched text that correspond to portions of the regular expression. To create tokens, enclose part of the regular expression in parentheses.

For example, this expression finds a date of the form `dd-mmm-yyyy`, including tokens for the day, month, and year.

```
str = 'Here is a date: 01-Apr-2020';
expression = '(\d+)-(\w+)-(\d+)';

mydate = regexp(str,expression,'tokens');
mydate{:}

ans =
    '01'    'April'    '2020'
```

You can associate names with tokens so that they are more easily identifiable:

```
str = 'Here is a date: 01-Apr-2020';
expression = '(?<day>\d+)-(?<month>\w+)-(?<year>\d+)';

mydate = regexp(str,expression,'names')

mydate =
      day: '01'
    month: 'Apr'
     year: '2020'
```

For more information, see “Tokens in Regular Expressions”.

See Also

`regexp` | `regexprep` | `regexptranslate` | `strfind` | `strjoin` | `strsplit` | `strrep`

Concepts

- “Lookahead Assertions in Regular Expressions”
- “Dynamic Regular Expressions”

regexprep

Purpose

Replace string using regular expression

Syntax

```
newStr = regexprep(str,expression,replace)
newStr =
regexprep(str,expression,replace,option1,...optionM)
```

Description

`newStr = regexprep(str,expression,replace)` replaces the text in `str` that matches `expression` with the text described by `replace`. The `regexprep` function returns the updated text in `newStr`.

- If `str` is a string, then `newStr` is also a string, even when `expression` or `replace` is a cell array of strings. When `expression` is a cell array, `regexprep` applies the first expression to the string, and then applies each subsequent expression to the preceding result.
- If `str` is a cell array, then `newStr` is a cell array with the same dimensions as `str`. For each element of `str`, the `regexprep` function applies each expression in sequence.
- If there are no matches to `expression`, then `newStr` is equivalent to `str`.

`newStr =`
`regexprep(str,expression,replace,option1,...optionM)` modifies the search using the specified options. For example, specify `'ignorecase'` to perform a case-insensitive match.

Input Arguments

str - Text to update

string | cell array of strings

Text to update, specified as a string or a cell array of strings.

Data Types

`char | cell`

expression - Regular expression

string | cell array of strings

Regular expression, specified as a string or a cell array of strings. Each expression can contain characters, metacharacters, operators, tokens, and flags that specify patterns to match in `str`.

The following tables describe the elements of regular expressions.

Metacharacters

Metacharacters represent letters, letter ranges, digits, and space characters. Use them to construct a generalized pattern of characters.

Metacharacter	Description	Example
`.`	Any single character, including white space	`'..ain'` matches sequences of five consecutive characters that end with `'ain'`.
[$c_1c_2c_3$]	Any character contained within the brackets	`'[rp]ain'` matches `'rain'` or `'pain'`.
[^$c_1c_2c_3$]	Any character not contained within the brackets	`'[^rp]ain'` matches all four-letter sequences that end in `'ain'`, except `'rain'` and `'pain'`. For example, it matches `'gain'`, `'lain'`, or `'vain'`.
[c_1-c_2]	Any character in the range of c_1 through c_2	`'[A-G]'` matches a single character in the range of `A` through `G`.
`\w`	Any alphabetic, numeric, or underscore character. For English character sets, `\w` is equivalent to `[a-zA-Z_0-9]`	`'\w*'` identifies a word.
`\W`	Any character that is not alphabetic, numeric, or underscore. For English character sets, `\W` is equivalent to `[^a-zA-Z_0-9]`	`'\W*'` identifies a substring that is not a word.

Metacharacter	Description	Example
\s	Any white-space character; equivalent to [\f\n\r\t\v]	'\w*n\s' matches words that end with the letter n, followed by a white-space character.
\S	Any non-white-space character; equivalent to [^ \f\n\r\t\v]	'\d\S' matches a numeric digit followed by any non-white-space character.
\d	Any numeric digit; equivalent to [0-9]	'\d*' matches any number of consecutive digits.
\D	Any nondigit character; equivalent to [^0-9]	'\w*\D\>' matches words that do not end with a numeric digit.
\oN or \o{N}	Character of octal value N	'\o{40}' matches the space character, defined by octal 40.
\xN or \x{N}	Character of hexadecimal value N	'\x2C' matches the comma character, defined by hex 2C.

Character Representation

Operator	Description
\a	Alarm (beep)
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
char	Any character with special meaning in regular expressions that you want to match literally (for example, use \\ to match a single backslash)

Quantifiers

Quantifiers specify the number of times a string pattern must occur in the matching string.

Quantifier	Matches the expression when it occurs...	Example
expr*	0 or more times consecutively.	'\w*' matches a word of any length.
expr?	0 times or 1 time.	'\w*(\.m)?' matches words that optionally end with the extension .m.
expr+	1 or more times consecutively.	'<img src="\w+\.gif">' matches an <img> HTML tag when the file name contains one or more characters.
expr{m,n}	At least m times, but no more than n times consecutively. {0,1} is equivalent to ?.	'\S{4,8}' matches between four and eight non-white-space characters.
expr{m,}	At least m times consecutively. {0,} and {1,} are equivalent to * and +, respectively.	'<a href="\w{1,}\.html">' matches an <a> HTML tag when the file name contains one or more characters.
expr{n}	Exactly n times consecutively. Equivalent to {n,n}.	'\d{4}' matches four consecutive digits.

Quantifiers can appear in three modes, described in the following table. *q* represents any of the quantifiers in the previous table.

Mode	Description	Example
expr*q*	Greedy expression: match as many characters as possible.	Given the string '<tr><td> text </td>', the expression '</?t.*>' matches all characters between <tr and /td>: '<tr><td> text </td>'
expr*q*?	Lazy expression: match as few characters as necessary.	Given the string '<tr><td> text </td>', the expression '</?t.*?>' ends each match at the first occurrence of the closing bracket (>): '<tr>' '<td>' '</td>'
expr*q*+	Possessive expression: match as much as possible, but do not rescan any portions of the string.	Given the string '<tr><td> text </td>', the expression '</?t.*+>' does not return any matches, because the closing bracket is captured using .*, and is not rescanned.

Grouping Operators

Grouping operators allow you to capture tokens, apply one operator to multiple elements, or disable backtracking in a specific group.

Grouping Operator	Description	Example
(expr)	Group elements of the expression and capture tokens.	'Joh?n\s(\w*)' captures a token that contains the last name of any person with the first name John or Jon.
(?:expr)	Group, but do not capture tokens.	'(?:[aeiou][^aeiou]){2}' matches two consecutive patterns of a vowel followed by a nonvowel, such as 'anon'. Without grouping, '[aeiou][^aeiou]{2}'matches a vowel followed by two nonvowels.
(?>expr)	Group atomically. Do not backtrack within the group to complete the match, and do not capture tokens.	'A(?>.*)Z' does not match 'AtoZ', although 'A(?:.*)Z' does. Using the atomic group, Z is captured using .* and is not rescanned.
(expr1|expr2)	Match expression expr1 or expression expr2. If there is a match with expr1, then expr2 is ignored. You can include ?: or ?> after the opening parenthesis to suppress tokens or group atomically.	'(let|tel)\w+' matches words in a string that start with let or tel.

Anchors

Anchors in the expression match the beginning or end of the string or word.

Anchor	Matches the...	Example
^expr	Beginning of the input string.	'^M\w*' matches a word starting with M at the beginning of the string.
expr$	End of the input string.	'\w*m$' matches words ending with m at the end of the string.
\<expr	Beginning of a word.	'\<n\w*' matches any words starting with n.
expr\>	End of a word.	'\w*e\>' matches any words ending with e.

Lookaround Assertions

Lookaround assertions look for string patterns that immediately precede or follow the intended match, but are not part of the match.

The pointer remains at the current location, and characters that correspond to the test expression are not captured or discarded. Therefore, lookahead assertions can match overlapping character groups.

Lookaround Assertion	Description	Example
expr(?=test)	Look ahead for characters that match test.	'\w*(?=ing)' matches strings that are followed by ing, such as 'Fly' and 'fall' in the input string 'Flying, not falling.'
expr(?!test)	Look ahead for characters that do not match test.	'i(?!ng)' matches instances of the letter i that are not followed by ng.

Lookaround Assertion	Description	Example
`(?<=test)expr`	Look behind for characters that match `test`..	`'(?<=re)\w*'` matches strings that follow `'re'`, such as `'new'`, `'use'`, and `'cycle'` in the input string `'renew, reuse, recycle'`
`(?<!test)expr`	Look behind for characters that do not match `test`.	`'(?<!\d)(\d)(?!\d)'` matches single-digit numbers (digits that do not precede or follow other digits).

If you specify a lookahead assertion *before* an expression, the operation is equivalent to a logical `AND`.

Operation	Description	Example
`(?=test)expr`	Match both `test` and `expr`.	`'(?=[a-z])[^aeiou]'` matches consonants.
`(?!test)expr`	Match `expr` and do not match `test`.	`'(?![aeiou])[a-z]'` matches consonants.

Logical and Conditional Operators

Logical and conditional operators allow you to test the state of a given condition, and then use the outcome to determine which string, if any, to match next. These operators support logical `OR`, and `if` or `if/else` conditions.

Conditions can be tokens, lookaround operators, or dynamic expressions of the form `(?@cmd)`. Dynamic expressions must return a logical or numeric value.

<table>
<tr><th>Conditional Operator</th><th>Description</th><th>Example</th></tr>
<tr><td>expr1|expr2</td><td>Match expression expr1 or expression expr2.

If there is a match with expr1, then expr2 is ignored.</td><td>'(let|tel)\w+' matches words in a string that start with let or tel.</td></tr>
<tr><td>(?(cond)expr)</td><td>If condition cond is true, then match expr.</td><td>'(?(?@ispc)[A-Z]:\\)' matches a drive name, such as C:\, when run on a Windows system.</td></tr>
<tr><td>(?(cond)expr1|expr2)</td><td>If condition cond is true, then match expr1. Otherwise, match expr2.</td><td>'Mr(s?)\..*?(?(1)her|his) \w*' matches strings that include her when the string begins with Mrs, or that include his when the string begins with Mr.</td></tr>
</table>

Token Operators

Tokens are portions of the matched text that you define by enclosing part of the regular expression in parentheses. You can refer to a token by its sequence in the string (an ordinal token), or assign names to tokens for easier code maintenance and readable output.

<table>
<tr><th>Ordinal Token Operator</th><th>Description</th><th>Example</th></tr>
<tr><td>(expr)</td><td>Capture in a token the characters that match the enclosed expression.</td><td>'Joh?n\s(\w*)' captures a token that contains the last name of any person with the first name John or Jon.</td></tr>
<tr><td>\N</td><td>Match the Nth token.</td><td>'<(\w+).*>.*</\1>' captures tokens for HTML tags, such</td></tr>
</table>

<table>
<tr><th>Ordinal Token Operator</th><th>Description</th><th>Example</th></tr>
<tr><td></td><td></td><td>as 'title' from the string '<title>Some text</title>'.</td></tr>
<tr><td>(?(N)expr1|expr2)</td><td>If the Nth token is found, then match expr1. Otherwise, match expr2.</td><td>'Mr(s?)\..*?(?(1)her|his) \w*' matches strings that include her when the string begins with Mrs, or that include his when the string begins with Mr.</td></tr>
</table>

<table>
<tr><th>Named Token Operator</th><th>Description</th><th>Example</th></tr>
<tr><td>(?<name>expr)</td><td>Capture in a named token the characters that match the enclosed expression.</td><td>'(?<month>\d+)-(?<day>\d+)-(?<yr>\d+)' creates named tokens for the month, day, and year in an input date string of the form mm-dd-yy.</td></tr>
<tr><td>\k<name></td><td>Match the token referred to by name.</td><td>'<(?<tag>\w+).*>.*</\k<tag>>' captures tokens for HTML tags, such as 'title' from the string '<title>Some text</title>'.</td></tr>
<tr><td>(?(name)expr1|expr2)</td><td>If the named token is found, then match expr1. Otherwise, match expr2.</td><td>'Mr(?<sex>s?)\..*?(?(sex)her|his) \w*' matches strings that include her when the string begins with Mrs, or that include his when the string begins with Mr.</td></tr>
</table>

Note If an expression has nested parentheses, MATLAB captures tokens that correspond to the outermost set of parentheses. For example, given the search pattern `'(and(y|rew))'`, MATLAB creates a token for `'andrew'` but not for `'y'` or `'rew'`.

Dynamic Regular Expressions

Dynamic expressions allow you to execute a MATLAB command or a regular expression to determine the text to match.

The parentheses that enclose dynamic expressions do *not* create a capturing group.

Operator	Description	Example
`(??expr)`	Parse `expr` and include the resulting string in the match expression. When parsed, `expr` must correspond to a complete, valid regular expression. Dynamic expressions that use the backslash escape character (\) require two backslashes: one for the initial parsing of `expr`, and one for the complete match.	`'^(\d+)((??\\w{$1}))'` determines how many characters to match by reading a digit at the beginning of the string. The dynamic expression is enclosed in a second set of parentheses so that the resulting match is captured in a token. For instance, matching `'5XXXXX'` captures tokens for `'5'` and `'XXXXX'`.
`(??@cmd)`	Execute the MATLAB command represented by `cmd`, and include the string returned by the command in the match expression.	`'(.{2,}).?(??@fliplr($1))'` finds palindromes that are at least four characters long, such as `'abba'`.
`(?@cmd)`	Execute the MATLAB command represented by `cmd`, but discard any output the command returns. (Helpful for diagnosing regular expressions.)	`'\w*?(\w)(?@disp($1))\1\w*'` matches words that include double letters (such as `pp`), and displays intermediate results.

Within dynamic expressions, use the following operators to define replacement strings.

Replacement String Operator	Description
$& or $0	Portion of the input string that is currently a match
$`	Portion of the input string that precedes the current match
$'	Portion of the input string that follows the current match (use $'' to represent the string $')
$N	Nth token
$<name>	Named token
${cmd}	String returned when MATLAB executes the command, cmd

Comments

Characters	Description	Example
(?#comment)	Insert a comment in the regular expression. The comment text is ignored when matching the input string.	'(?# Initial digit)\<\d\w+' includes a comment, and matches words that begin with a number.

Search Flags

Search flags modify the behavior for matching expressions. An alternative to using a search flag within an expression is to pass an option input argument.

Flag	Description
`(?-i)`	Match letter case (default for `regexp` and `regexprep`).
`(?i)`	Do not match letter case (default for `regexpi`).
`(?s)`	Match dot (`.`) in the pattern string with any character (default).
`(?-s)`	Match dot in the pattern with any character that is not a newline character.
`(?-m)`	Match the `^` and `$` metacharacters at the beginning and end of a string (default).
`(?m)`	Match the `^` and `$` metacharacters at the beginning and end of a line.
`(?-x)`	Include space characters and comments when matching (default).
`(?x)`	Ignore space characters and comments when matching. Use `'\ '` and `'\#'` to match space and `#` characters.

The expression that the flag modifies can appear either after the parentheses, such as

```
(?i)\w*
```

or inside the parentheses and separated from the flag with a colon (`:`), such as

```
(?i:\w*)
```

The latter syntax allows you to change the behavior for part of a larger expression.

Data Types
`char | cell`

replace - Replacement text
string | cell array of strings

Replacement text, specified as a string or a cell array of strings, as follows:

- If `replace` is a single string and `expression` is a cell array of strings, then `regexprep` uses the same replacement text for each expression.
- If `replace` is a cell array of `N` strings and `expression` is a single string, then `regexprep` attempts `N` matches and replacements.
- If both `replace` and `expression` are cell arrays of strings, then they must contain the same number of elements. `regexprep` pairs each `replace` element with its matching element in `expression`.

The replacement text can include regular characters, special characters (such as tabs or new lines), or string operators, as shown in the following tables.

Replacement String Operator	Description
`$&` or `$0`	Portion of the input string that is currently a match
`` $` ``	Portion of the input string that precedes the current match
`$'`	Portion of the input string that follows the current match (use `$''` to represent the string `$'`)
`$N`	`N`th token
`$<name>`	Named token
`${cmd}`	String returned when MATLAB executes the command, `cmd`

Operator	Description
`\a`	Alarm (beep)
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\char`	Any character with special meaning in regular expressions that you want to match literally (for example, use `\\` to match a single backslash)

Data Types
`char | cell`

option - Search or replacement option

`'once' | N | 'warnings' | 'ignorecase' | 'preservecase' | 'emptymatch' | 'dotexceptnewline' | 'lineanchors'`

Search or replacement option, specified as a string or an integer value, as shown in the following table.

Options come in sets: one option that corresponds to the default behavior, and one or two options that allow you to override the default. Specify only one option from a set. Options can appear in any order.

Default	Override	Description
`'all'`	`'once'`	Match and replace the expression as many times as possible (default), or only once.
	`N`	Replace only the `N`th occurrence of the match, where `N` is an integer value.

Default	Override	Description
'nowarnings'	'warnings'	Suppress warnings (default), or display them.
'matchcase'	'ignorecase'	Match letter case (default), or ignore case while matching and replacing.
	'preservecase'	Ignore case while matching, but preserve the case of corresponding characters in the original string while replacing.
'noemptymatch'	'emptymatch'	Ignore zero length matches (default), or include them.
'dotall'	'dotexceptnewline'	Match dot with any character (default), or all except newline (`\n`).
'stringanchors'	'lineanchors'	Apply `^` and `$` metacharacters to the beginning and end of a string (default), or to the beginning and end of a line.
'literalspacing'	'freespacing'	Include space characters and comments when matching (default), or ignore them. With `freespacing`, use `'\ '` and `'\#'` to match space and `#` characters.

Data Types
`char`

Output Arguments

newStr - Updated text

string | cell array of strings

Updated text, returned as a string or a cell array of strings. The data type of `newStr` is the same as the data type of `str`.

Examples

Update a Single String

Replace words that begin with `M`, end with `y`, and have at least one character between them.

```
str = 'My flowers may bloom in May';
expression = 'M(\w+)y';
```

```
replace = 'April';

newStr = regexprep(str,expression,replace)

newStr =

My flowers may bloom in April
```

Include Tokens in Replacement Text

Replace variations of the phrase `'walk up'` by capturing the letters that follow `'walk'` in a token.

```
str = 'I walk up, they walked up, we are walking up.';
expression = 'walk(\w*) up';
replace = 'ascend$1';

newStr = regexprep(str,expression,replace)

newStr =

I ascend, they ascended, we are ascending.
```

Include Dynamic Expression in Replacement Text

Replace lowercase letters at the beginning of sentences with their uppercase equivalents using the `upper` function.

```
str = 'here are two sentences. neither is capitalized.';
expression = '(^|\.)\s*.';
replace = '${upper($0)}';

newStr = regexprep(str,expression,replace)

newStr =

Here are two sentences. Neither is capitalized.
```

The regular expression matches single characters (`.`) that follow the beginning of the string (`^`) or a period (`\.`) and any whitespace (`\s*`).

The `replace` expression calls the `upper` function for the currently matching character (`$0`).

Update Multiple Strings

Replace each occurrence of a double letter in a set of strings with the symbols `'--'`.

```
str = {                                  ...
'Whose woods these are I think I know.' ; ...
'His house is in the village though;'   ; ...
'He will not see me stopping here'      ; ...
'To watch his woods fill up with snow.'};

expression = '(.)\1';
replace = '--';
newStr = regexprep(str,expression,replace)

newStr =

    'Whose w--ds these are I think I know.'
    'His house is in the vi--age though;'
    'He wi-- not s-- me sto--ing here'
    'To watch his w--ds fi-- up with snow.'
```

Preserve Case in Original String

Ignore letter case in the regular expression when finding matches, but mimic the letter case of the original string when updating.

```
str = 'My flowers may bloom in May';
expression = 'M(\w+)y';
replace = 'April';

newStr = regexprep(str,expression,replace,'preservecase')

newStr =

My flowers april bloom in April
```

Replace Zero-Length Matches

Insert text at the beginning of a string using the `'^'` operator, which returns a zero-length match, and the `'emptymatch'` keyword.

```
str = 'abc';
expression = '^';
replace = '__';

newStr = regexprep(str,expression,replace,'emptymatch')

newStr =

__abc
```

See Also

`regexp` | `strcmp` | `strfind` | `strrep`

Concepts

- "Lookahead Assertions in Regular Expressions"
- "Tokens in Regular Expressions"
- "Dynamic Regular Expressions"

Purpose

Translate string into regular expression

Syntax

```
s2 = regexptranslate(type, s1)
```

Description

`s2 = regexptranslate(type, s1)` translates string `s1` into a regular expression string `s2` that you can then use as input into one of the MATLAB regular expression functions such as `regexp`. The `type` input can be either one of the following strings that define the type of translation to be performed. See "Regular Expressions" in the MATLAB Programming Fundamentals documentation for more information.

Type	Description
`'escape'`	Translate all special characters (e.g., '$', '.', '?', '[') in string `s1` so that they are treated as literal characters when used in the `regexp` and `regexprep` functions. The translation inserts an escape character ('\') before each special character in `s1`. Return the new string in `s2`.
`'wildcard'`	Translate all wildcard and '.' characters in string `s1` so that they are treated as literal wildcards and periods when used in the `regexp` and `regexprep` functions. The translation replaces all instances of '*' with '.*', all instances of '?' with '.', and all instances of '.' with '\.'. Return the new string in `s2`.

Examples

Example 1 — Using the 'escape' Option

Because `regexp` interprets the sequence '\n' as a newline character, it cannot locate the two consecutive characters '\' and 'n' in this string:

```
str = 'The sequence \n generates a new line';
pat = '\n';

regexp(str, pat)
ans =
     []
```

To have regexp interpret the expression expr as the characters '\' and 'n', first translate the expression using regexptranslate:

```
pat2 = regexptranslate('escape', pat)
pat2 =
   \\n

regexp(str, pat2)
ans =
    14
```

Example 2 — Using 'escape' In a Replacement String

Replace the word 'walk' with 'ascend' in this string, treating the characters '$1' as a token designator:

```
str = 'I walk up, they walked up, we are walking up.';
pat = 'walk(\w*) up';

regexprep(str, pat, 'ascend$1')
ans =
   I ascend, they ascended, we are ascending.
```

Make another replacement on the same string, this time treating the '$1' as literal characters:

```
regexprep(str, pat, regexptranslate('escape', 'ascend$1'))
ans =
   I ascend$1, they ascend$1, we are ascend$1.
```

Example 3 — Using the 'wildcard' Option

Given the following string of filenames, pick out just the MAT-files. Use regexptranslate to interpret the '*' wildcard as '\w+' instead of as a regular expression quantifier:

```
files = ['test1.mat, myfile.mat, newfile.txt, ' ...
         'jan30.mat, table3.xls'];
regexp(str, regexptranslate('wildcard', '*.mat'), 'match')
ans =
```

```
   'test1.mat' 'myfile.mat' 'jan30.mat'
```

To see the translation, you can type

```
regexptranslate('wildcard','*.mat')
ans =
   \w+\.mat
```

See Also

regexp | regexpi | regexprep

How To

- “Regular Expressions”

Purpose

Associate event handler for COM object event at run time

Syntax

```
h.registerevent(eventhandler)
registerevent(h, eventhandler)
```

Description

`h.registerevent(eventhandler)` registers event handler routines with their corresponding events. The `eventhandler` argument can be either a string that specifies the name of the event handler function, or a function handle that maps to that function. Strings used in the `eventhandler` argument are not case sensitive.

`registerevent(h, eventhandler)` is an alternate syntax.

COM functions are available on Microsoft Windows systems only.

Examples

Show events in the MATLAB sample control:

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.2', [0 0 200 200], f);
h.events
```

MATLAB displays all events associated with the instance of the control (output is formatted):

```
Click = void Click()
DblClick = void DblClick()
MouseDown = void MouseDown(int16 Button, int16 Shift,
  Variant x, Variant y)
Event_Args = void Event_Args(int16 typeshort,
  int32 typelong, double typedouble, string typestring,
  bool typebool)
```

Register all events with the same event handler routine, `sampev`:

```
h.registerevent('sampev');
h.eventlisteners
```

MATLAB displays:

```
ans =
    'Click'         'sampev'
    'DblClick'      'sampev'
    'MouseDown'     'sampev'
    'Event_Args'    'sampev'
```

Register individual events:

```
%Unregister existing events
h.unregisterallevents;
%Register specific events
h.registerevent({'click' 'myclick'; ...
    'dblclick' 'my2click'});
h.eventlisteners
```

MATLAB displays:

```
ans =
    'click'       'myclick'
    'dblclick'    'my2click'
```

Register events using a function handle (`@sampev`) instead of the function name:

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

See Also

`events (COM)` | `eventlisteners` | `unregisterevent` | `unregisterallevents` | `isevent`

How To

- "Writing Event Handlers"

rehash

Purpose

Refresh function and file system path caches

Syntax

```
rehash
rehash path
rehash toolbox
rehash pathreset
rehash toolboxreset
rehash toolboxcache
```

Description

`rehash` with no arguments updates the MATLAB list of known files and classes for directories on the search path that are not in *`matlabroot`*`/toolbox`. It compares the timestamps for loaded functions against their timestamps on disk. It clears loaded functions if the files on disk are newer. All of this normally happens each time MATLAB displays the Command Window prompt. Use `rehash` with no arguments only when you run a program file that updates another program file, and the calling file needs to reuse the updated version of the second file before the calling file has finished running.

`rehash` **`path`** performs the same updates as `rehash`, but uses a different technique for detecting the files and directories that require updates. Run `rehash path` only if you receive a warning during MATLAB startup notifying you that MATLAB could not tell if a directory has changed, and you encounter problems with MATLAB not using the most current versions of your program files.

`rehash` **`toolbox`** performs the same updates as rehash path, except it updates the list of known files and classes for *all* directories on the search path, including those in *`matlabroot`*`/toolbox`. Run `rehash toolbox` when you change, add, or remove files in *`matlabroot`*`/toolbox` during a session. Typically, you should not make changes to files and directories in *`matlabroot`*`/toolbox`.

`rehash` **`pathreset`** performs the same updates as `rehash` **`path`**, and also ensures the known files and classes list follows precedence rules for shadowed functions.

`rehash` **`toolboxreset`** performs the same updates as `rehash` **`toolbox`**, and also ensures the known files and classes list follows precedence rules for shadowed functions.

`rehash` **`toolboxcache`** performs the same updates as `rehash` **`toolbox`**, and also updates the cache file. This is the equivalent of clicking the **Update Toolbox Path Cache** button in the General Preferences dialog box.

See Also

`addpath` | `clear` | `matlabroot` | `path` | `rmpath`

How To

- "Toolbox Path Caching in MATLAB"
- "What Is the MATLAB Search Path?"

Purpose

Release COM interface

Syntax

```
h.release
release(h)
```

Description

`h.release` releases the interface and all resources used by the interface. You must release the handle when you are done with the interface. A released interface is no longer valid. MATLAB generates an error if you try to use an object that represents that interface.

`release(h)` is an alternate syntax.

Releasing the interface does not delete the control itself (see the `delete` function), since other interfaces on that object might still be active.

COM functions are available on Microsoft Windows systems only.

See Also

`delete (COM)` | `actxcontrol` | `actxserver`

How To

- Releasing Interfaces

Purpose

Equality and sorting of handle objects

Syntax

```
TF = eq(H1,H2)
TF = ne(H1,H2)
TF = lt(H1,H2)
TF = le(H1,H2)
TF = gt(H1,H2)
TF = ge(H1,H2)
```

Description

`TF = eq(H1,H2)`

`TF = ne(H1,H2)`

`TF = lt(H1,H2)`

`TF = le(H1,H2)`

`TF = gt(H1,H2)`

`TF = ge(H1,H2)`

For each pair of input arrays (`H1` and `H2`), a logical array of the same size is returned in which each element is an element-wise equality or comparison test result. These methods perform scalar expansion in the same way as the MATLAB built-in functions. See `relationaloperators` for more information.

You can make the following assumptions about the result of a handle comparison:

- The same two handles always compare as equal and the repeated comparison of any two handles always yields the same result in the same MATLAB session.
- Different handles are always not-equal.
- The order of handle values is purely arbitrary and has no connection to the state of the handle objects being compared.
- If the input arrays belong to different classes (including the case where one input array belongs to a non-handle class such as `double`) then the comparison is always false.

- If a comparison is made between a handle object and an object of a dominant class, the method of the dominant class is invoked. You should generally test only like objects because a dominant class might not define one of these methods.
- An error occurs if the input arrays are not the same size and neither is scalar.

Use `isequal` when you want to determine if handle objects with different handles have the same data in all object properties.

See Also

`handle` | `meta.class`

Purpose

Remainder after division

Syntax

```
R = rem(X,Y)
```

Description

`R = rem(X,Y)` if `Y ~= 0`, returns `X - n.*Y` where `n = fix(X./Y)`. If `Y` is not an integer and the quotient `X./Y` is within roundoff error of an integer, then `n` is that integer. The inputs `X` and `Y` must be real arrays of the same size, or real scalars.

The following are true by convention:

- `rem(X,0)` is `NaN`
- `rem(X,X)` for `X~=0` is `0`
- `rem(X,Y)` for `X~=Y` and `Y~=0` has the same sign as `X`.

Tips

`mod(X,Y)` for `X~=Y` and `Y~=0` has the same sign as `Y`.

`rem(X,Y)` and `mod(X,Y)` are equal if `X` and `Y` have the same sign, but differ by `Y` if `X` and `Y` have different signs.

The `rem` function returns a result that is between 0 and `sign(X)*abs(Y)`. If `Y` is zero, `rem` returns `NaN`.

See Also

`mod`

containers.Map.remove

Purpose

Remove key-value pairs from `containers.Map` object

Syntax

`remove(mapObj,keySet)`

Description

`remove(mapObj,keySet)` erases all specified keys, and the values associated with them, from `mapObj`. Input `keySet` can be a scalar key or a cell array of keys.

Input Arguments

mapObj

Object of class `containers.Map`.

keySet

Scalar value, string, or cell array that specifies keys in `mapObj` to delete.

Examples

Remove Key-Value Pairs from a Map

Create a map and view the keys and the `Count` property:

```
myKeys = {'a','b','c','d'};
myValues = [1,2,3,4];
mapObj = containers.Map(myKeys,myValues);

mapKeys = keys(mapObj)
mapCount = mapObj.Count
```

The initial map contains four key-value pairs:

```
mapKeys =
    'a'    'b'    'c'    'd'

mapCount =
                     4
```

Remove the pairs corresponding to keys `b` and `d`:

```
keySet = {'b','d'};
```

```
remove(mapObj,keySet)

mapKeys = keys(mapObj)
mapCount = mapObj.Count
```

The modified map contains two key-value pairs:

```
mapKeys =
    'a'    'c'

mapCount =
                    2
```

See Also

containers.Map | keys | values | isKey

removets

Purpose

Remove `timeseries` objects from `tscollection` object

Syntax

```
tsc = removets(tsc,Name)
```

Description

`tsc = removets(tsc,Name)` removes one or more `timeseries` objects with the name specified in `Name` from the `tscollection` object tsc. `Name` can either be a string or a cell array of strings.

Examples

The following example shows how to remove a time series from a `tscollection`.

1 Create two `timeseries` objects, `ts1` and `ts2`.

```
ts1=timeseries([1.1 2.9 3.7 4.0 3.0],1:5,'name','acceleration');
ts2=timeseries([3.2 4.2 6.2 8.5 1.1],1:5,'name','speed');
```

2 Create a `tscollection` object `tsc`, which includes `ts1` and `ts2`.

```
tsc=tscollection({ts1 ts2});
```

3 To view the members of `tsc`, type the following at the MATLAB prompt:

```
tsc
```

The response is

```
Time Series Collection Object: unnamed

Time vector characteristics

      Start time            1 seconds
      End time              5 seconds

Member Time Series Objects:
```

```
      acceleration
      speed
```

The members of `tsc` are listed by name at the bottom: `acceleration` and `speed`. These are the `Name` properties of `ts1` and `ts2`, respectively.

4 Remove `ts2` from `tsc`.

```
tsc=removets(tsc,'speed');
```

5 To view the current members of `tsc`, type the following at the MATLAB prompt:

```
tsc
```

The response is

```
Time Series Collection Object: unnamed

Time vector characteristics

      Start time            1 seconds
      End time              5 seconds

Member Time Series Objects:
      acceleration
```

The remaining member of `tsc` is `acceleration`. The `timeseries` `speed` has been removed.

See Also

`addts` | `tscollection`

FTP.rename

Purpose

Rename file on FTP server

Syntax

`rename(ftpobj,oldname,newname)`

Description

`rename(ftpobj,oldname,newname)` changes the name of the file `oldname` to `newname` in the current folder on an FTP server.

Input Arguments

ftpobj

FTP object created by `ftp`.

oldname

String enclosed in single quotation marks that specifies the original name of the file.

newname

String enclosed in single quotation marks that specifies a new name for the file.

Examples

Suppose that hypothetical FTP server `ftp.testsite.com` contains a file named `testfile.m`. Rename the file to `final.m`:

```
test = ftp('ftp.testsite.com');
rename(test, 'testfile.m', 'final.m');
close(test);
```

See Also

`delete` | `dir` | `ftp` | `mget` | `mput`

Purpose

Replicate and tile array

Syntax

```
B = repmat(A,m,n)
B = repmat(A,[m n])
B = repmat(A,[m n p...])
```

Description

`B = repmat(A,m,n)` creates a large matrix `B` consisting of an `m`-by-`n` tiling of copies of `A`. The size of `B` is `[size(A,1)*m, (size(A,2)*n]`. The statement `repmat(A,n)` creates an `n`-by-`n` tiling.

`B = repmat(A,[m n])` accomplishes the same result as `repmat(A,m,n)`.

`B = repmat(A,[m n p...])` produces a multidimensional array `B` composed of copies of `A`. The size of `B` is `[size(A,1)*m, size(A,2)*n, size(A,3)*p, ...]`.

Tips

`repmat(A,m,n)`, when `A` is a scalar, produces an `m`-by-`n` matrix filled with `A`'s value and having `A`'s `class`. For certain values, you can achieve the same results using other functions, as shown by the following examples:

- `repmat(NaN,m,n)` returns the same result as `NaN(m,n)`.
- `repmat(single(inf),m,n)` is the same as `inf(m,n,'single')`.
- `repmat(int8(0),m,n)` is the same as `zeros(m,n,'int8')`.
- `repmat(uint32(1),m,n)` is the same as `ones(m,n,'uint32')`.
- `repmat(eps,m,n)` is the same as `eps(ones(m,n))`.

Examples

Example 1

In this example, `repmat` replicates 12 copies of the second-order identity matrix, resulting in a “checkerboard” pattern.

```
B = repmat(eye(2),3,4)

B =
     1     0     1     0     1     0     1     0
     0     1     0     1     0     1     0     1
     1     0     1     0     1     0     1     0
```

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

The statement `N = repmat(NaN,[2 3])` creates a 2-by-3 matrix of `NaN`s.

Example 2

If you have code that uses `repmat` and also a binary operator or function, you can transform the code to use the `bsxfun` function instead. In certain cases, this can provide a simpler and faster solution.

This example replaces the sum of two repmat operations with a single call to `bsxfun`:

```
x = 1:5; y = (1:10)';

% Replace this code
repmat(x,10,1) + repmat(y,1,5)

% with the following:
bsxfun(@plus, x, y)
```

See Also

`reshape` | `bsxfun` | `NaN` | `Inf` | `ones` | `zeros`

Purpose

Select or interpolate data in `tscollection` using new time vector

Syntax

```
tsc = resample(tsc,Time)
tsc = resample(tsc,Time,interp_method)
tsc = resample(tsc,Time,interp_method,code)
```

Description

`tsc = resample(tsc,Time)` resamples the `tscollection` object tsc on the new Time vector. When `tsc` uses date strings and Time is numeric, Time is treated as numerical specified relative to the `tsc.TimeInfo.StartDate` property and in the same units that tsc uses. The `resample` method uses the default interpolation method for each time series member.

`tsc = resample(tsc,Time,interp_method)` resamples the `tscollection` object tsc using the interpolation method given by the string interp_method. Valid interpolation methods include `'linear'` and `'zoh'` (zero-order hold).

`tsc = resample(tsc,Time,interp_method,code)` resamples the `tscollection` object tsc using the interpolation method given by the string interp_method. The integer code is a user-defined quality code for resampling, applied to all samples.

Examples

The following example shows how to resample a `tscollection` that consists of two `timeseries` members.

1 Create two `timeseries` objects.

```
ts1=timeseries([1.1 2.9 3.7 4.0 3.0],1:5,'name','acceleration');
ts2=timeseries([3.2 4.2 6.2 8.5 1.1],1:5,'name','speed');
```

2 Create a `tscollection` `tsc`.

```
tsc=tscollection({ts1 ts2});
```

The time vector of the collection `tsc` is [1:5], which is the same as for `ts1` and `ts2` (individually).

3 Get the interpolation method for `acceleration` by typing

```
tsc.acceleration
```

MATLAB responds with

```
Time Series Object: acceleration

Time vector characteristics

      Length                  5
      Start time              1 seconds
      End time                5 seconds

Data characteristics

      Interpolation method  linear
      Size                  [1  1  5]
      Data type             double
```

4 Set the interpolation method for `speed` to zero-order hold by typing

```
setinterpmethod(tsc.speed,'zoh')
```

MATLAB responds with

```
Time Series Object: acceleration

Time vector characteristics

      Length                  5
      Start time              1 seconds
      End time                5 seconds
```

```
Data characteristics

      Interpolation method  zoh
      Size                  [1  1  5]
      Data type             double
```

5. Resample the time-series collection `tsc` by individually resampling each time-series member of the collection and using its interpolation method.

```
res_tsc=resample(tsc,[1 1.5 3.5 4.5 4.9])
```

See Also

`getinterpmethod` | `setinterpmethod` | `tscollection`

reset

Purpose

Reset graphics object properties to their defaults

Syntax

```
reset(h)
```

Description

`reset(h)` resets all properties having factory defaults on the object identified by `h`. To see the list of factory defaults, use the statement

```
get(0,'factory')
```

If `h` is a `figure`, the MATLAB software does not reset `Position`, `Units`, `WindowStyle`, or `PaperUnits`. If `h` is an `axes`, MATLAB does not reset `Position` and `Units`.

Examples

`reset(gca)` resets the properties of the current axes.

`reset(gcf)` resets the properties of the current figure.

See Also

`cla` | `clf` | `gca` | `gcf` | `hold`

Purpose

Reset random number stream

Class

`RandStream`

Syntax

```
reset(s)
reset(s,seed)
```

Description

`reset(s)` resets the generator for the random stream, `s`, to the internal state corresponding to its seed. This is similar to clearing `s` and recreating it using `RandStream(Type,...)`, except that `reset` does not set the stream's `NormalTransform`, `Antithetic`, and `FullPrecision` properties to their original values.

`reset(s,seed)` resets the generator for the random stream, `s`, to the internal state corresponding to `seed` (the seed value), and it updates the `seed` property of `s`. The value of `seed` must be an integer between 0 and $2^{32} - 1$. Resetting a stream's seed can invalidate independence with other streams.

Note Resetting a stream should be used primarily for reproducing results.

Examples

Example 1

Reset a random number stream to its initial state. This does not create a random number stream, it simply resets the stream:

```
stream = RandStream('twister','Seed',0)

stream =

mt19937ar random stream
             Seed: 0
  NormalTransform: Ziggurat
```

```
reset(stream);
stream.Seed

ans =

           0
```

Example 2

Reset a random number stream using a specific seed:

```
stream = RandStream('twister','Seed',0)

stream =

mt19937ar random stream
             Seed: 0
  NormalTransform: Ziggurat

reset(stream,1);
stream.Seed

ans =

           1
```

See Also

`RandStream` | `RandStream.getGlobalStream`

Purpose

Reshape array

Syntax

```
B = reshape(A,m,n)
B = reshape(A,m,n,p,...)
B = reshape(A,[m n p ...])
B = reshape(A,...,[],...)
B = reshape(A,siz)
```

Description

`B = reshape(A,m,n)` returns the `m`-by-`n` matrix `B` whose elements are taken column-wise from `A`. An error results if `A` does not have `m*n` elements.

`B = reshape(A,m,n,p,...)` or `B = reshape(A,[m n p ...])` returns an n-dimensional array with the same elements as `A` but reshaped to have the size `m`-by-`n`-by-`p`-by-`...`. The product of the specified dimensions, `m*n*p*`..., must be the same as `numel(A)`.

`B = reshape(A,...,[],...)` calculates the length of the dimension represented by the placeholder `[]`, such that the product of the dimensions equals `numel(A)`. The value of `numel(A)` must be evenly divisible by the product of the specified dimensions. You can use only one occurrence of `[]`.

`B = reshape(A,siz)` returns an n-dimensional array with the same elements as `A`, but reshaped to `siz`, a vector representing the dimensions of the reshaped array. The quantity `prod(siz)` must be the same as `numel(A)`.

Examples

Reshape a 3-by-4 matrix into a 2-by-6 matrix.

```
A =
     1     4     7    10
     2     5     8    11
     3     6     9    12

B = reshape(A,2,6)

B =
```

```
     1     3     5     7     9    11
     2     4     6     8    10    12
B = reshape(A,2,[])

B =
     1     3     5     7     9    11
     2     4     6     8    10    12
```

See Also

shiftdim | squeeze | circshift | permute | repmat | colon (:)

Purpose

Convert between partial fraction expansion and polynomial coefficients

Syntax

```
[r,p,k] = residue(b,a)
[b,a] = residue(r,p,k)
```

Description

The `residue` function converts a quotient of polynomials to pole-residue representation, and back again.

`[r,p,k] = residue(b,a)` finds the residues, poles, and direct term of a partial fraction expansion of the ratio of two polynomials, $b(s)$ and $a(s)$, of the form

$$\frac{b(s)}{a(s)} = \frac{b_1 s^m + b_2 s^{m-1} + b_3 s^{m-2} + \ldots + b_{m+1}}{a_1 s^n + a_2 s^{n-1} + a_3 s^{n-2} + \ldots + a_{n+1}}$$

where b_j and a_j are the `j`th elements of the input vectors `b` and `a`.

`[b,a] = residue(r,p,k)` converts the partial fraction expansion back to the polynomials with coefficients in `b` and `a`.

Definitions

If there are no multiple roots, then

$$\frac{b(s)}{a(s)} = \frac{r_1}{s - p_1} + \frac{r_2}{s - p_2} + \ldots + \frac{r_n}{s - p_n} + k(s)$$

The number of poles `n` is

```
n = length(a)-1 = length(r) = length(p)
```

The direct term coefficient vector is empty if `length(b) < length(a)`; otherwise

```
length(k) = length(b)-length(a)+1
```

If `p(j) = ...  = p(j+m-1)` is a pole of multiplicity `m`, then the expansion includes terms of the form

$$\frac{r_j}{s-p_j}+\frac{r_{j+1}}{(s-p_j)^2}+\ldots+\frac{r_{j+m-1}}{(s-p_j)^m}$$

Arguments

`b,a`	Vectors that specify the coefficients of the polynomials in descending powers of s
`r`	Column vector of residues
`p`	Column vector of poles
`k`	Row vector of direct terms

Algorithms

It first obtains the poles with `roots`. Next, if the fraction is nonproper, the direct term `k` is found using `deconv`, which performs polynomial long division. Finally, the residues are determined by evaluating the polynomial with individual roots removed. For repeated roots, `resi2` computes the residues at the repeated root locations.

Limitations

Numerically, the partial fraction expansion of a ratio of polynomials represents an ill-posed problem. If the denominator polynomial, $a(s)$, is near a polynomial with multiple roots, then small changes in the data, including roundoff errors, can make arbitrarily large changes in the resulting poles and residues. Problem formulations making use of state-space or zero-pole representations are preferable.

Examples

If the ratio of two polynomials is expressed as

$$\frac{b(s)}{a(s)}=\frac{5s^3+3s^2-2s+7}{-4s^3+8s+3}$$

then

```
b = [ 5 3 -2 7]
a = [-4 0 8 3]
```

and you can calculate the partial fraction expansion as

```
[r, p, k] = residue(b,a)

r =
   -1.4167
   -0.6653
    1.3320

p =
    1.5737
   -1.1644
   -0.4093

k =
   -1.2500
```

Now, convert the partial fraction expansion back to polynomial coefficients.

```
[b,a] = residue(r,p,k)

b =
   -1.2500   -0.7500    0.5000   -1.7500

a =
    1.0000   -0.0000   -2.0000   -0.7500
```

The result can be expressed as

$$\frac{b(s)}{a(s)} = \frac{-1.25s^3 - 0.75s^2 + 0.50s - 1.75}{s^3 - 2.00s - 0.75}.$$

Note that the result is normalized for the leading coefficient in the denominator.

References

[1] Oppenheim, A.V. and R.W. Schafer, *Digital Signal Processing*, Prentice-Hall, 1975, p. 56.

See Also

`deconv` | `poly` | `roots`

restoredefaultpath

Purpose

Restore default search path

Alternatives

As an alternative to the `restoredefaultpath` function, use the Set Path dialog box.

Syntax

```
restoredefaultpath
restoredefaultpath; matlabrc
```

Description

`restoredefaultpath` sets the search path to include only folders for MathWorks installed products. Use `restoredefaultpath` when you are having problems with the search path.

`restoredefaultpath; matlabrc` sets the search path to include only folders for MathWorks installed products and corrects search path problems encountered during startup.

MATLAB does not support issuing `restoredefaultpath` from a UNC path name. Doing so might result in MATLAB being unable to find files on the search path. If you do issue `restoredefaultpath` from a UNC path name, restore the expected behavior by changing the current folder to an absolute path, and then reissuing `restoredefaultpath`.

See Also

`addpath` | `genpath` | `matlabrc` | `rmpath` | `savepath`

How To

- "Path Unsuccessfully Set at Startup"
- "What Is the MATLAB Search Path?"

rethrow

Purpose

Reissue error

Note As of version 7.5, MATLAB supports error handling that is based on the `MException` class. Calling `rethrow` with a structure argument, as described on this page, is now replaced by calling rethrow with an `MException` object, as described on the reference page for `rethrow(MException)`. `rethrow` called with a structure input will be removed in a future version.

Syntax

```
rethrow(errorStruct)
```

Description

`rethrow(errorStruct)` reissues the error specified by `errorStruct`. The currently running function terminates and control returns to the keyboard (or to any enclosing `catch` block). The `errorStruct` argument must be a MATLAB structure containing at least the `message` and `identifier` fields:

Fieldname	Description
`message`	Text of the error message
`identifier`	Message identifier of the error message
`stack`	Information about the error from the program stack

See "Message Identifiers" in the MATLAB documentation for more information on the syntax and usage of message identifiers.

Tips

The `errorStruct` input can contain the field `stack`, identical in format to the output of the `dbstack` command. If the `stack` field is present, the stack of the rethrown error will be set to that value. Otherwise, the stack will be set to the line at which the rethrow occurs.

Examples

`rethrow` is usually used in conjunction with `try`-`catch` statements to reissue an error from a `catch` block after performing `catch`-related operations. For example,

```
try
   do_something
catch
   do_cleanup
   rethrow(previous_error)
end
```

See Also

`rethrow(MException)` | `throw(MException)` | `throwAsCaller(MException)` | `try` | `catch` | `error` | `assert` | `dbstop`

rethrow (MException)

Purpose

Reissue existing exception

Syntax

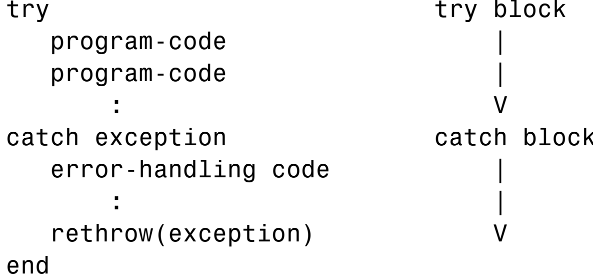
`rethrow(exception)`

Description

`rethrow(exception)` forces an exception (i.e., error report) to be reissued by MATLAB after the error reporting process has been temporarily suspended to diagnose or remedy the problem. MATLAB typically responds to errors by terminating the currently running program. Errors occurring within a try block, however, bypass this mechanism and transfer control of the program to error handling code in the `catch` block instead. This enables you to write your own error handling procedures for parts of your program that require them.

The *exception* input is a scalar object of the `MException` class that contains information about the cause and location of the error.

The code segment below shows the format of a typical `try/catch` statement.



```
try                                 try block
   program-code                          |
   program-code                          |
        :                                V
catch exception                     catch block
   error-handling code                   |
        :                                |
   rethrow(exception)                    V
end
```

An error detected within the `try` block causes MATLAB to enter the corresponding `catch` block. The error record constructed by MATLAB in the process of reporting this error passes to the `catch` command in the statement

```
catch exception
```

Error handling code within the `catch` block uses the information in the error record to address the problem in some predefined manner. The

catch block shown here ends with a rethrow statement which throws the exception returned in the catch statement, and then terminates the function:

```
rethrow(exception)
```

The most significant difference between rethrow and other MATLAB functions that throw exceptions is in how rethrow handles a piece of the exception record called the *stack*. The stack keeps a record of where the error occurred and what functions were called in the process. It is a struct array composed of the following fields, where each element of the array represents an exception:

Fields of the Exception Stack	Description
line	Line number from which the exception was thrown.
name	Name of the function being executed at the time.
file	Name of the file containing that function.

Functions such as error, assert, or throw, create the stack with the location from which they were executed. Calling rethrow, however, preserves information from the original exception. In doing so, rethrow enables you to retrace the path taken to the source of the error.

Tips

There are four ways to throw an exception in MATLAB (see the list below). Use the first of these when testing the outcome of some action for failure and reporting the failure to MATLAB. Use one of the remaining techniques to throw an existing exception.

- Test the result of some action taken by your program. If the result is found to be incorrect or unexpected, compose an appropriate message and message identifier, and pass these to MATLAB using the error function.

- Reissue the original exception by throwing the initial error record unmodified. Use the MException rethrow method to do this.
- Collect additional information on the cause of the error, store it in a new or modified error record, and issue a new exception based on that record. Use the MException addCause and throw methods to do this.
- Make it appear that the error originated in the caller of the currently running function. Use the MException throwAsCaller method to do this.

rethrow can only issue a previously caught exception. Calling rethrow on an exception that was not previously thrown is an error.

Examples

This example shows the difference between using throw and rethrow at the end of a catch block. The combineArrays function vertically concatenates arrays A and B. When the two arrays have rows of unequal length, the function throws an error.

The first time you run the function, comment out the rethrow command at the end of the catch block so that the function calls throw instead:

```
function C = combineArrays(A, B)
try
    catAlongDim1(A, B);              % Line 3
catch exception
    throw(exception)                 % Line 5
    % rethrow(exception)             % Line 6
end

function catAlongDim1(V1, V2)
    C = cat(1, V1, V2);              % Line 10
```

When MATLAB throws the exception, it reports an error on line 5 which is the line that calls throw. In some cases, that might be what you want but, in this case, it does not show the true source of the error.

```
A = 4:3:19;     B = 3:4:19;
combineArrays(A, B)
```

```
Error using combineArrays (line 5)
CAT arguments dimensions are not consistent.
```

Make the following changes to `combineArrays.m` so that you use `rethrow` instead:

```
% throw(exception)                      % Line 5
rethrow(exception)                      % Line 6
```

Run the function again. This time, line 10 is the first line reported which is where the MATLAB concatenation function `cat` was called and the exception originated. The next error reported is on line 3 which is where the call to `catAlongDim1` was called:

```
combineArrays(A, B)

Error using cat
CAT arguments dimensions are not consistent.

Error in combineArrays>catAlongDim1 (line 10)
    C = cat(1, V1, V2);                 % Line 10
Error in combineArrays (line 3)
    catAlongDim1(A, B);                 % Line 3
```

See Also

`try` | `catch` | `error` | `assert` | `MException` | `throw(MException)` | `throwAsCaller(MException)` | `addCause(MException)` | `getReport(MException)` | `last(MException)`

return

Purpose

Return to invoking function

Syntax

```
return
```

Description

`return` causes a normal return to the invoking function or to the keyboard. It also terminates keyboard mode.

Examples

This determinant function uses `return` to handle the special case of an empty matrix:

```
function d = det(A)
%DET det(A) is the determinant of A.
if isempty(A)
    d = 1;
    return
else
    ...
end
```

See Also

`break` | `continue` | `disp` | `end` | `error` | `for` | `if` | `keyboard` | `switch` | `while`

Purpose

Write modified metadata to existing IFD

Syntax

`tiffobj.rewriteDirectory()`

Description

`tiffobj.rewriteDirectory()` writes modified metadata (tag) data to an existing directory. Use this tag when you want to change the value of a tag in an existing image file directory.

Examples

Open a `Tiff` object for modification and modify the value of a tag. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path.

```
t = Tiff('myfile.tif', 'r');
%  Modify the value of a tag.
t.setTag('Software','MATLAB');
t.rewriteDirectory();
```

References

This method corresponds to the `TIFFRewriteDirectory` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff.writeDirectory`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

rgb2hsv

Purpose

Convert RGB colormap to HSV colormap

Syntax

```
cmap = rgb2hsv(M)
hsv_image = rgb2hsv(rgb_image)
```

Description

`cmap = rgb2hsv(M)` converts an RGB colormap `M` to an HSV colormap `cmap`. Both colormaps are *m*-by-3 matrices. The elements of both colormaps are in the range 0 to 1.

The columns of the input matrix `M` represent intensities of red, green, and blue, respectively. The columns of the output matrix `cmap` represent hue, saturation, and value, respectively.

`hsv_image = rgb2hsv(rgb_image)` converts the RGB image to the equivalent HSV image. RGB is an m-by-n-by-3 image array whose three planes contain the red, green, and blue components for the image. HSV is returned as an m-by-n-by-3 image array whose three planes contain the hue, saturation, and value components for the image.

See Also

`brighten` | `colormap` | `hsv2rgb` | `rgbplot`

Purpose

Convert RGB image to indexed image

Syntax

```
[X,map] = rgb2ind(RGB,n)
X = rgb2ind(RGB, map)
[X,map] = rgb2ind(RGB, tol)
[ ___ ] = rgb2ind( ___ ,dither_option)
```

Description

`[X,map] = rgb2ind(RGB,n)` converts the RGB image to an indexed image `X` using minimum variance quantization and dithering. `map` contains at most `n` colors. `n` must be less than or equal to 65,536.

`X = rgb2ind(RGB, map)` converts the RGB image to an indexed image `X` with colormap `map` using the inverse colormap algorithm. `size(map,1)` must be less than or equal to 65,536.

`[X,map] = rgb2ind(RGB, tol)` converts the RGB image to an indexed image `X` using uniform quantization and dithering. `map` contains at most `(floor(1/tol)+1)^3` colors. `tol` must be between 0.0 and 1.0.

`[ ___ ] = rgb2ind( ___ ,dither_option)` enables or disables dithering. `dither_option` is a string that can have one of these values.

`'dither'` (default)	dithers, if necessary, to achieve better color resolution at the expense of spatial resolution.
`'nodither'`	maps each color in the original image to the closest color in the new map. No dithering is performed.

Note The values in the resultant image `X` are indexes into the colormap `map` and should not be used in mathematical processing, such as filtering operations.

Class Support

The input image can be of class `uint8`, `uint16`, `single`, or `double`. If the length of `map` is less than or equal to 256, the output image is of class `uint8`. Otherwise, the output image is of class `uint16`.

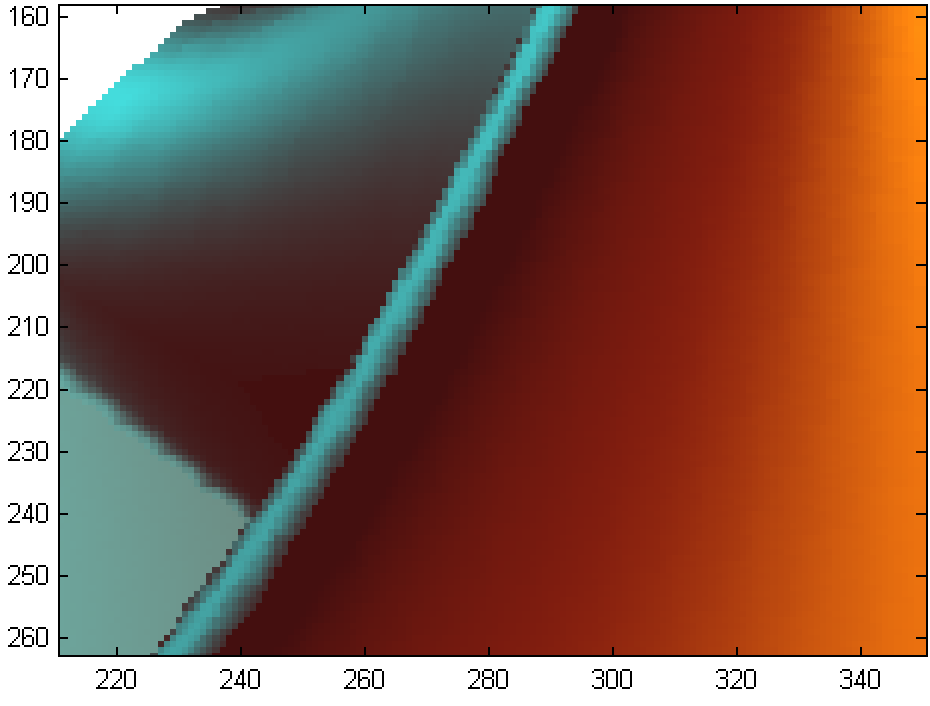
Algorithms

- **Uniform Quantization** — Uniform quantization cuts the RGB color cube into smaller cubes of length tol. [insert material marked "A" on PDF.]
- **Minimum Variance Quantization** — Minimum variance quantization cuts the RGB color cube into smaller boxes (not necessarily cubes) of different sizes, depending on how the colors are distributed in the image. If the input image actually uses fewer colors than the number specified, the output colormap is also smaller.
- **Inverse Colormap** — The inverse colormap algorithm quantizes the specified colormap into 32 distinct levels per color component. Then, for each pixel in the input image, the closest color in the quantized colormap is found.

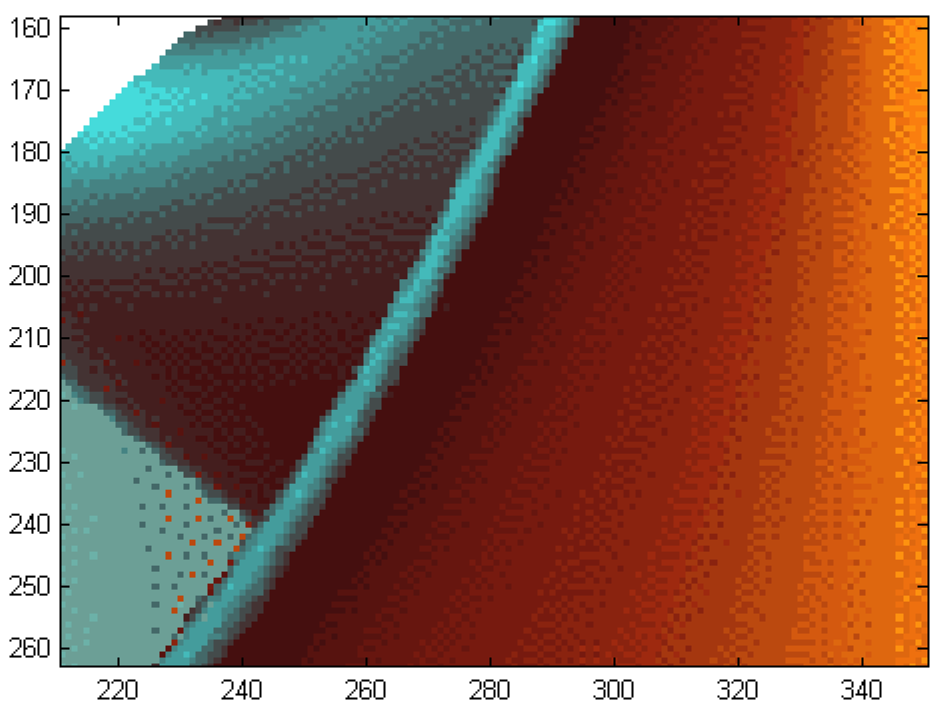
Examples

Read and display a truecolor `uint8` PNG image of the MathWorks logo. Zoom in to see color variation better.

```
imfile = fullfile(matlabroot,...
         'toolbox','matlab','demos','html','logodemo_01.png');
RGB = imread(imfile,'PNG');
figure('Name','RGB Truecolor Image')
imagesc(RGB)
axis image
zoom(4)
```




```
% Convert RGB to an indexed image with 32 colors
[IND,map] = rgb2ind(RGB,32);
figure('Name','Indexed image with 32 Colors')
imagesc(IND)
colormap(map)
axis image
zoom(4)
```



References

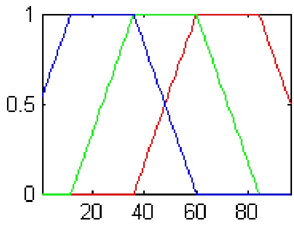
[1] Spencer W. Thomas, "Efficient Inverse Color Map Computation", *Graphics Gems II*, (ed. James Arvo), Academic Press: Boston. 1991. (includes source code)

See Also

`cmunique` | `dither` | `imapprox` | `ind2rgb`

Purpose

Plot colormap



Syntax

```
rgbplot(cmap)
```

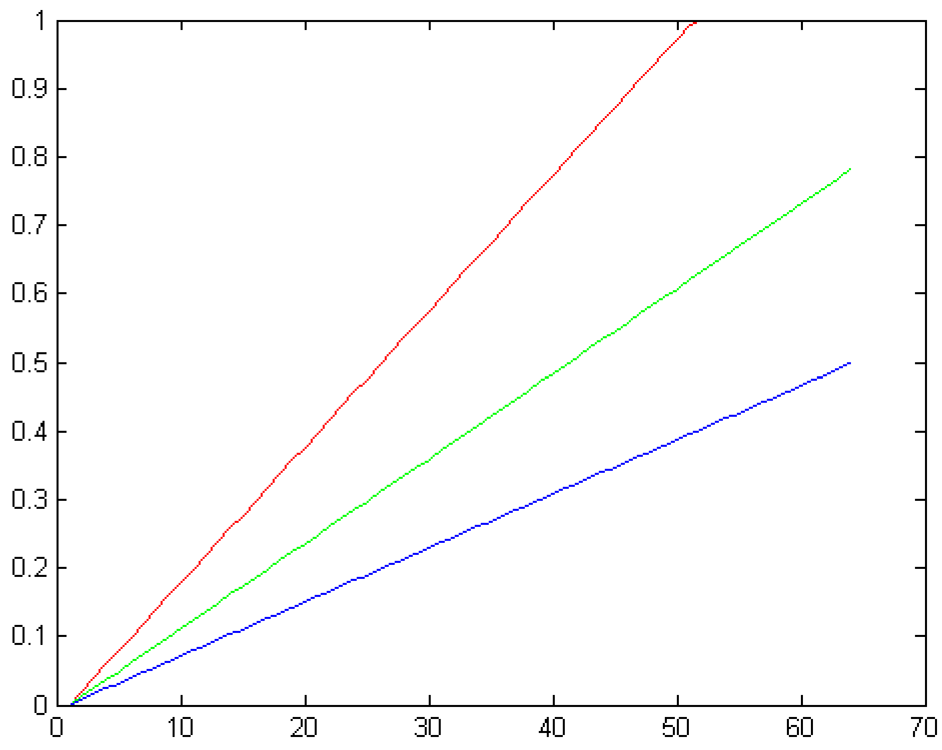
Description

`rgbplot(cmap)` plots the three columns of `cmap`, where `cmap` is an *m*-by-3 colormap matrix. `rgbplot` draws the first column in red, the second in green, and the third in blue.

Examples

Plot the RGB values of the `copper` colormap.

```
rgbplot(copper)
```

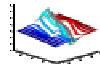


See Also

`colormap`

Purpose

Ribbon plot



Syntax

```
ribbon(Y)
ribbon(X,Y)
ribbon(X,Y,width)
ribbon(axes_handle,...)
h = ribbon(...)
```

Description

`ribbon(Y)` plots the columns of `Y` as three-dimensional ribbons of uniform width using `X = 1:size(Y,1)`. Ribbons advance along the *x*-axis centered on tick marks at unit intervals, three-quarters of a unit in width. Ribbon maps values in `X` to colors in colormap linearly. To change ribbon colors in the graph, change the `colormap`.

`ribbon(X,Y)` plots three dimensional ribbons for data in `Y`, centered at locations specified in `X`. `X` and `Y` are vectors or matrices of the same size. Additionally, `X` can be a row or a column vector, and `Y` a matrix with `length(X)` rows. When `Y` is a matrix, `ribbon` plots each column in `Y` as a ribbon at the corresponding `X` location.

`ribbon(X,Y,width)` specifies the width of the ribbons. The default is `0.75`. If `width = 1`, the ribbons touch, leaving no space between them when viewed down the *z*-axis. If `width > 1`, ribbons overlap and can intersect.

`ribbon(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

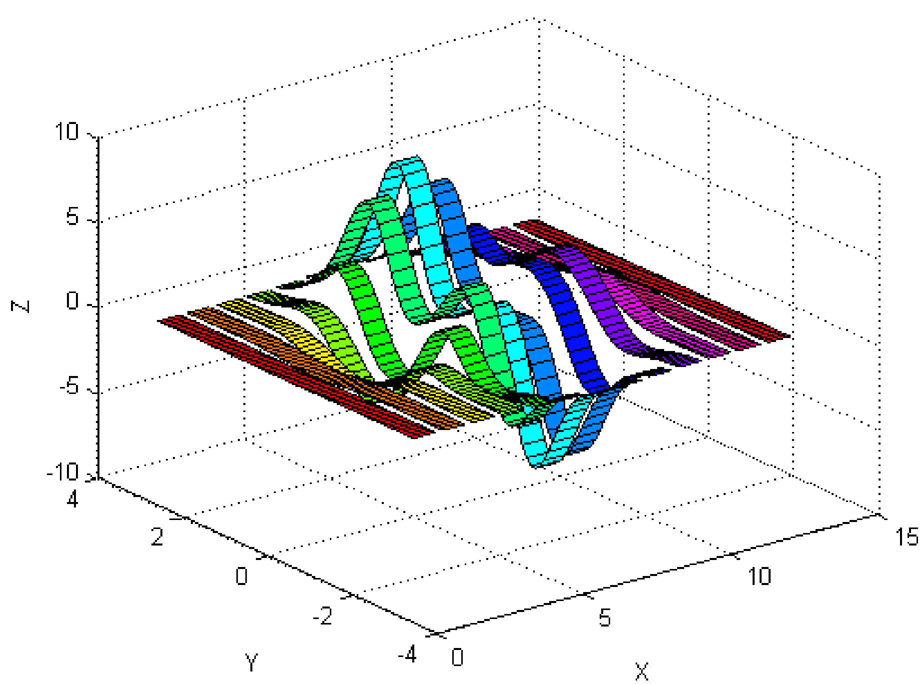
`h = ribbon(...)` returns a vector of handles to surface graphics objects. `ribbon` returns one handle per strip.

Examples

Create a ribbon plot of the `peaks` function.

```
[x,y] = meshgrid(-3:.5:3,-3:.1:3);
z = peaks(x,y);
```

```
ribbon(y,z)
xlabel('X')
ylabel('Y')
zlabel('Z')
colormap hsv
```



See Also

plot | plot3 | surface | waterfall

Purpose

Remove application-defined data

Syntax

```
rmappdata(h,name)
```

Description

`rmappdata(h,name)` removes the application-defined data `name` from the object specified by handle `h`.

Tips

Application data is data that is meaningful to or defined by your application which you attach to a figure or any GUI component (other than ActiveX controls) through its `AppData` property. Only Handle Graphics MATLAB objects use this property.

See Also

`getappdata` | `isappdata` | `setappdata`

rmdir

Purpose

Remove folder

Syntax

```
rmdir(folderName)
rmdir(folderName,'s')
[status, message, messageid] = rmdir(folderName,'s')
```

Description

`rmdir(folderName)` removes the folder `folderName` from the current folder if `folderName` is empty. If `folderName` is not in the current folder, then specify the relative path or the full path for `folderName`.

`rmdir(folderName,'s')` removes the folder `folderName` and its contents from the current folder. With the `'s'` option, `rmdir` attempts to remove all subfolders and files in `folderName` regardless of their write permissions.

`[status, message, messageid] = rmdir(folderName,'s')` removes the folder `folderName` and its contents from the current folder, returning the status, a message, and the MATLAB message ID.

Tips

- If you specify the `'s'` flag or include a wildcard in the folder name, MATLAB produces an error if it is unable to remove all folders. The error message lists the folder and files that MATLAB could not remove.

Input Arguments

folderName

String specifying the absolute or relative path name of the folder you want to remove.

Default: None

's'

Literal string that directs `rmdir` to remove all subfolders and files in the specified folder, regardless of their write permissions. The result for read-only files follows the practices of the operating system.

Default: `rmdir` does not remove subfolders and files in the specified folder.

Output Arguments

status

Logical scalar indicating the outcome of the `rmdir` operation. The status value is `1` if the operation was successful and `0` if it returned an error.

message

String containing the warning or error message text if the operation is unsuccessful. An empty string, if the operation is successful.

messageid

String containing the warning or error message ID, if the operation is unsuccessful. MATLAB returns an empty string if the operation is successful.

Examples

These examples remove an empty folder, `myfiles`, assuming it is in the current folder:

```
% Remove myfiles from the current folder:

rmdir('myfiles')

% Use the relative path to remove myfiles. Assuming
% the current folder is matlab/work and myfiles is in
% d:/matlab/work/project, type this:

rmdir('project/myfiles')

% Use the full path to remove myfiles, assuming
% the current folder is matlab/work and myfiles is in
% d:/matlab/work/project:

rmdir('d:/matlab/work/project/myfiles')
```

This example removes the `myfiles` folder and its contents, assuming `myfiles` is in the current folder:

```
rmdir('myfiles','s')
```

This example unsuccessfully attempts to remove the `myfiles` folder and its contents. It directs MATLAB to display the results.

```
[stat, mess, id]=rmdir('myfiles')
```

MATLAB returns:

```
stat =
     0

mess =

No directories were removed.

id =

MATLAB:RMDIR:NoDirectoriesRemoved
```

This example successfully removes the `myfiles` folder and its contents. It directs MATLAB to display the results.

```
[stat, mess]=rmdir('myfiles','s')
```

MATLAB returns:

```
stat =
     1

mess =
```

''

Alternatives

Open the Current Folder browser by running `filebrowser`. Then, in the Current Folder browser, right-click the folder name and select **Delete** from the context menu.

See Also

| `cd` | `copyfile` | `delete` | `dir` | `fileattrib` | `filebrowser` | `mkdir` | `movefile`

FTP.rmdir

Purpose

Remove folder on FTP server

Syntax

```
rmdir(ftpobj,folder)
```

Description

`rmdir(ftpobj,folder)` removes the specified folder from the current folder on an FTP server.

Input Arguments

ftpobj

FTP object created by `ftp`.

folder

String enclosed in single quotation marks that specifies the name of the folder to delete.

Examples

Remove the folder `temp` from the hypothetical FTP server `ftp.testsite.com`:

```
test = ftp('ftp.testsite.com');
rmdir(test, 'temp');
```

See Also

`cd` | `delete` | `dir` | `ftp` | `mkdir`

Purpose

Remove fields from structure

Syntax

```
s = rmfield(s,field)
```

Description

`s = rmfield(s,field)` removes the specified field or fields from structure array `s`. Specify multiple fields using a cell array of strings. The dimensions of `s` remain the same.

Input Arguments

s - Input structure

structure array

Input structure, specified as a structure array.

Data Types
`struct`

field - Field name or names

character array | cell array of strings

Field name or names, specified as a character array or a cell array of strings.

Example: `'f1'`

Example: `{'f1';'f2'}`

Data Types
`char | cell`

Examples

Remove Single Field

Define a scalar structure with fields named `a`, `b`, and `c`.

```
s.a = 1;
s.b = 2;
s.c = 3;
```

Remove field `b`.

```
field = 'b';
```

```
s = rmfield(s,field)
```

```
s =
    a: 1
    c: 3
```

Remove Multiple Fields

Define a scalar structure with fields `first`, `second`, `third`, and `fourth`.

```
S.first = 1;
S.second = 2;
S.third = 3;
S.fourth = 4;
```

Remove fields `first` and `fourth`.

```
fields = {'first','fourth'};
S = rmfield(S,fields)
```

```
S =
    second: 2
     third: 3
```

See Also

`fieldnames` | `isfield` | `orderfields`

Concepts

- "Generate Field Names from Variables"

Purpose Remove folders from search path

Syntax

```
rmpath(folderName)
```

Description `rmpath(folderName)` removes the specified folder from the search path.

Input Arguments

folderName - Name of folder
string

Name of folder to remove from the search path, specified as a string. Use the full path name for `folderName`.

Example: `'c:\matlab\work'`

Example: `'/home/user/matlab'`

Examples

Remove Folder from Search Path

Remove `/usr/local/matlab/mytools` from the search path.

```
rmpath('/usr/local/matlab/mytools')
```

See Also `addpath | savepath | path`

Concepts

- "What Is the MATLAB Search Path?"

rmpref

Purpose

Remove preference

Syntax

```
rmpref('group','pref')
rmpref('group',{'pref1','pref2',...'prefn'})
rmpref('group')
```

Description

`rmpref('group','pref')` removes the preference specified by `group` and `pref`. It is an error to remove a preference that does not exist.

`rmpref('group',{'pref1','pref2',...'prefn'})` removes each preference specified in the cell array of preference names. It is an error if any of the preferences do not exist.

`rmpref('group')` removes all the preferences for the specified `group`. It is an error to remove a group that does not exist.

Examples

```
addpref('mytoolbox','version','1.0')
rmpref('mytoolbox')
```

See Also

`addpref` | `getpref` | `ispref` | `setpref` | `uigetpref` | `uisetpref`

Purpose

Control random number generation

Syntax

```
rng(sd)
rng('shuffle')
rng(sd, generator)
rng('shuffle', generator)
rng('default')
scurr = rng
rng(s)
sprev = rng(...)
```

Description

Note To use the `rng` function instead of `rand` or `randn` with the `'seed'`, `'state'`, or `'twister'` inputs, see the documentation on "Updating Your Random Number Generator Syntax"

`rng(sd)` seeds the random number generator using the nonnegative integer `sd` so that `rand`, `randi`, and `randn` produce a predictable sequence of numbers.

`rng('shuffle')` seeds the random number generator based on the current time so that `rand`, `randi`, and `randn` produce a different sequence of numbers after each time you call `rng`.

`rng(sd, generator)` and `rng('shuffle', generator)` additionally specify the type of the random number generator used by `rand`, `randi`, and `randn`. The `generator` input is one of:

Generator	Description
`'twister'`	Mersenne Twister
`'combRecursive'`	Combined Multiple Recursive
`'multFibonacci'`	Multiplicative Lagged Fibonacci
`'v5uniform'`	Legacy MATLAB 5.0 uniform generator

Generator	Description
'v5normal'	Legacy MATLAB 5.0 normal generator
'v4'	Legacy MATLAB 4.0 generator

rng('default') puts the settings of the random number generator used by rand, randi, and randn to their default values so that they produce the same random numbers as if you restarted MATLAB. In this release, the default settings are the Mersenne Twister with seed 0.

scurr = rng returns the current settings of the random number generator used by rand, randi, and randn. The settings are returned in a structure scurr with fields 'Type', 'Seed', and 'State'.

rng(s) restores the settings of the random number generator used by rand, randi, and randn back to the values captured previously with a command such as s = rng.

sprev = rng(...) returns the previous settings of the random number generator used by rand, randi, and randn before changing the settings.

Examples

Example 1 — Retrieve and Restore Generator Settings

Save the current generator settings in s:

```
s = rng;
```

Call rand to generate a vector of random values:

```
x = rand(1,5)

x =

    0.8147    0.9058    0.1270    0.9134    0.6324
```

Restore the original generator settings by calling rng. Generate a new set of random values and verify that x and y are equal:

```
rng(s);
y = rand(1,5)
```

```
y =

    0.8147    0.9058    0.1270    0.9134    0.6324
```

Example 2 — Restore Settings for Legacy Generator

Use the legacy generator.

```
sprev = rng(0,'v5uniform')

sprev =
     Type: 'twister'
     Seed: 0
    State: [625x1 uint32]

x = rand

x =

    0.9501
```

Restore the previous settings by calling `rng`:

```
rng(sprev)
```

See Also

`rand` | `randi` | `randn` | `RandStream` | `now`

root object

Purpose

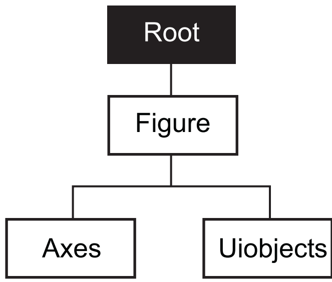
Root

Description

The root is a graphics object that corresponds to the computer screen. There is only one root object and it has no parent. The children of the root object are figures.

The root object exists when you start MATLAB; you never have to create it and you cannot destroy it. Use `set` and `get` to access the root properties.

Object Hierarchy



See Also

`diary` | `echo` | `figure` | `format` | `gcf` | `get` | `set` | Root Properties

Purpose

Root properties

Modifying Properties

You can set and query graphics object properties in two ways:

- Property Editor is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see "Setting Default Property Values".

Root Properties

This section provides a description of properties. Curly braces `{ }` enclose default values.

`BusyAction`

`cancel | {queue}`

Not used by the root object.

`ButtonDownFcn`

string

Not used by the root object.

`CallbackObject`

handle (read-only)

Handle of current callback's object. Contains the handle of the object whose callback routine is currently executing. If no callback routines are executing, this property contains an empty double array `[ ]`. See also the `gco` command.

`Children`

vector of handles

Handles of child objects. A vector containing the handles of all nonhidden figure objects (see `HandleVisibility` for more

information). You can change the order of the handles and thereby change the sorting order of the figures on the display.

`Clipping`

`{on} | off`

Clipping mode. This property has no effect on the root object.

`CommandWindowSize`

[columns rows]

Current size of command window. Size of the MATLAB command window, in a two-element vector. The first element is the number of columns wide and the second element is the number of rows tall.

For example, a value of `[50,25]` means that 50 characters can display across the Command Window, and 25 lines can display without scrolling.

Enabling the Command Window Display preference **Set matrix display width to eighty columns** forces the returned value for number of columns wide to be 80 regardless of the window width.

`CreateFcn`

The root object does not use this property.

`CurrentFigure`

figure handle

Handle of the current figure, which is the one most recently created, clicked in, or made current with the statement:

```
figure(h) % h is a figure handle
```

which re-sorts the figure to the top of the figures displayed on the screen, or:

```
set(0,'CurrentFigure',h)
```

which does not re-sort the figures. In these statements, `h` is the handle of an existing figure. If there are no figure objects:

```
get(0,'CurrentFigure')
```

returns an empty double array. Note, however, that `gcf` always returns a figure handle. Calling `gcf` creates a new figure if there are no existing figure objects that can become the current figure (for example, figures with hidden handles cannot become the current figure).

`DeleteFcn`
string

This property is not used because you cannot delete the root object.

`Diary`
`on | {off}`

Diary file mode. When this property is `on`, MATLAB maintains a file (whose name is specified by the `DiaryFile` property) that saves a copy of all keyboard input and most of the resulting output. See also the `diary` command.

`DiaryFile`
string

Diary file name. The default name is `diary`. Set this property to the name of the file MATLAB uses write the command-line diary when the `Diary` property is set to on.

`Echo`
`on | {off}`

Script echoing mode. When `Echo` is `on`, MATLAB displays each line of a script file as it executes. See also the `echo` command.

`ErrorMessage`
string

Text of last error message. Contains the last error message issued by MATLAB.

`FixedWidthFontName`
font name

Fixed-width font to use for axes, text, and uicontrols whose `FontName` *is set to* `FixedWidth`. MATLAB uses the font name specified for this property as the value for axes, text, and uicontrol `FontName` properties when their `FontName` property is `FixedWidth`.

Specifying the font name with this property eliminates the need to hardcode font names in MATLAB applications and thereby enables these applications to run without modification in locales where non-ASCII character sets are required. In these cases, MATLAB attempts to set the value of `FixedWidthFontName` to the correct value for a given locale.

MATLAB application developers should not change this property, but should create axes, text, and uicontrols with `FontName` properties set to `FixedWidth` when they want to use a fixed-width font for these objects.

MATLAB end users can set this property if they do not want to use the preselected value. `Courier` is the default in locales that use Latin-based characters.

`Format`
`short | {shortE} | long | longE | bank | hex | + | rat`

Output format mode. Format used to display numbers. See also the `format` command.

- `short` — Fixed-point format with 5 digits
- `shortE` — Floating-point format with 5 digits

- `shortG` — Fixed- or floating-point format displaying as many significant figures as possible with 5 digits
- `long` — Scaled fixed-point format with 15 digits
- `longE` — Floating-point format with 15 digits
- `longG` — Fixed- or floating-point format displaying as many significant figures as possible with 15 digits
- `bank` — Fixed-format of dollars and cents
- `hex` — Hexadecimal format
- `+` — Displays `+` and `-` symbols
- `rat` — Approximation by ratio of small integers

`FormatSpacing`
`compact | {loose}`

Output format spacing (see also `format` command).

- `compact` — Suppress extra line feeds for more compact display.
- `loose` — Display extra line feeds for a more readable display.

`HandleVisibility`
`{on} | callback | off`

This property is not useful on the root object because the handle is always `0`.

`HitTest`
`{on} | off`

This property is not useful on the root object.

`Interruptible`
`{on} | off`

This property is not useful on the root object.

`Language`
string

System environment setting.

`MonitorPositions`
Width and height of monitors, in pixels. Contains information about the size and relative location of monitors connected to your computer. The information returned by the `MonitorPosition` property depends on which computer system you are using.

Windows Systems

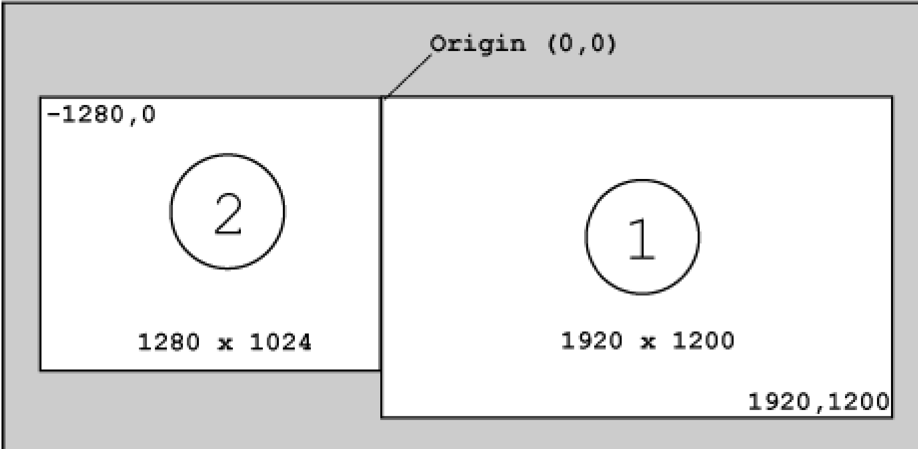
Windows systems define one monitor as device 1, which becomes the reference by which MATLAB determines other monitor positions. The position data is of the form:

```
[xmin2,ymin2,xmax2,ymax2;
 xmin1,ymin1,xmax1,ymax1]
```

The values for minimum and maximum are relative to the origin.

The following picture show the values returned when querying the `MonitorPositions` property on a Windows system having two monitors arranged as shown.

Windows System with Two Monitors




```
get(0,'MonitorPosition')

-1279     1       0       1024
    1     1    1920       1200
```

The `MonitorPositions` property contains an n-by-4 array, with each row representing a monitor position. The first row is the nth monitor and the last row corresponds to device `1`.

The monitor labeled as device `1` in the Windows control panel remains the reference monitor that defines the position of the origin however you reposition the monitors.

Linux Systems

On Linux systems, the upper-left corner of a rectangle enclosing the system monitors forms the origin. The position data is of the form:

```
[xP yP widthP heightP;
 xS yS widthS heightS]
```

Where the values represent the offset from the left (x), the offset from the top (y), and the width and height of the monitor. The `MonitorPositions` property contains an n-by-4 array, with each row representing a monitor position. The first row is the position of the primary monitor.

The following picture show the values returned when querying the `MonitorPositions` property on a Linux system that has two monitors arranged as shown.

Linux System with Two Monitors


```
get(0,'MonitorPosition')

1281     0       1920        1200
   1     503     1280        1024
```

MATLAB on Linux reads control panel settings only at startup. MATLAB does not recognize changes made to the arrangement of monitors during a MATLAB session.

Macintosh Systems

MATLAB on Macintosh systems recognize only the main monitor. The position data is of the form:

```
[x,y,width,height-menuHieght]
```

Where the values are `x = 0`, `y = 0`, monitor width, and monitor height minus the height of the menubar.

The main monitor is determined by which display has the menu bar. The **System Preferences Displays** control panel provides a way to arrange representations of multiple displays, and set one as the main monitor. To make another display the main monitor, drag the representation of the menu bar in the panel from the default display to the other display.

`Parent`

handle

Handle of parent object. This property always contains an empty double array, because the root object has no parent.

`PointerLocation`

`[x,y]`

Current location of pointer. A vector containing the *x*- and *y*-coordinates of the pointer position, measured from the lower left corner of the screen. Move the pointer by changing the values of this property. The `Units` property determines the units of this measurement.

`PointerLocation` always contains the current pointer location, even if the pointer is not in a MATLAB window. A callback

routine querying the `PointerLocation` property can get a value that is different from the location of the pointer when the callback was triggered. This difference results from delays in callback execution caused by competition for system resources.

On Apple Macintosh platforms, you cannot change the pointer location using the `set` command.

`PointerWindow`
handle (read-only)

Handle of window containing the pointer. MATLAB sets this property to the handle of the figure containing the pointer. If the pointer is not in a MATLAB window, the value of this property is `0`. A callback routine querying the `PointerWindow` can get the wrong window handle if you move the pointer to another window before the callback executes. This error results from delays in callback execution caused by competition for system resources.

`RecursionLimit`
integer

Number of nested MATLAB file calls. The limit of the number of nested calls to MATLAB files that MATLAB makes before stopping (or potentially running out of memory). By default the value is set to a large value. Setting this property to a smaller value (something like `150`, for example) should prevent MATLAB from running out of memory and instead causes MATLAB to issue an error when it reaches the limit.

`ScreenDepth`
bits per pixel

Screen depth. The depth of the display bitmap (the number of bits per pixel). The maximum number of simultaneously displayed colors on the current graphics device is 2 raised to this power.

`ScreenDepth` supersedes the `BlackAndWhite` property. To override automatic hardware checking, set this property to `1`. Then MATLAB assumes the display is monochrome. Setting `ScreenDepth` to `1` is useful if MATLAB is running on color hardware but is being displayed on a monochrome terminal. Such a situation can cause MATLAB to determine erroneously that the display is color.

`ScreenPixelsPerInch`
Display resolution

DPI setting for your display. Contains the setting of your display resolution specified in your system preferences.

`ScreenSize`
four-element rectangle vector (read-only)

Screen size. A four-element vector:

```
[left,bottom,width,height]
```

that defines the display size. `left` and `bottom` are `0` for all `Units` except `pixels`, in which case `left` and `bottom` are both `1`. `width` and `height` are the screen dimensions in units specified by the `Units` property.

Determining Screen Size

Note that the screen size in absolute units (for example, inches) is determined by dividing the number of pixels in width and height by the screen DPI (see the `ScreenPixelPerInch` property). This value is approximate and might not represent the actual size of the screen.

Note that the `ScreenSize` property is static. Its values are read-only at MATLAB startup and not updated if system display settings change. Also, the values returned might not represent

the usable screen size for application developers due to the presence of other GUIs, such as the Microsoft Windows task bar.

`Selected`
`on | off`

This property has no effect on the root object.

`SelectionHighlight`
`{on} | off`

This property has no effect on the root object.

`ShowHiddenHandles`
`on | {off}`

Show or hide handles marked as hidden. When set to `on`, this property disables handle hiding and exposes all object handles regardless of the setting of an object's `HandleVisibility` property. When set to `off`, all objects so marked remain hidden within the graphics hierarchy.

`Tag`
string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. While it is not necessary to identify the root object with a tag (since its handle is always `0`), you can use this property to store any string value, which you can later retrieve using `get`.

`Type`
string (read-only)

Class of graphics object. For the root object, `Type` is always the string `root`.

`UIContextMenu`
handle

This property is not used by the root object.

`Units`

`{pixels} | normalized | inches | centimeters | points | characters`

Unit of measurement. Specifies the units MATLAB uses to interpret size and location data contained in the `PointerLocation` and `ScreenSize` properties. All units are measured from the lower left corner of the screen.

- `normalized` — Map the lower left corner of the screen to `(0,0)` and the upper right corner to `(1.0,1.0)`.
- `inches`, `centimeters`, and `points` — Absolute units. 1 point = $^1/_{72}$ inch.
- `characters` — Based on the size of characters in the default system font. The width of one `characters` unit is the width of the letter `x`, and the height of one `characters` unit is the distance between the baselines of two lines of text.

If you change the value of `Units`, it is good practice to return it to its default value after completing your operation, so as not to affect other functions that assume `Units` is set to the default value.

`UserData`

matrix

User-specified data. Data you want to associate with the root object. The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`

`{on} | off`

Object visibility. This property has no effect on the root object.

See Also

`root object`

Purpose

Polynomial roots

Syntax

```
r = roots(c)
```

Description

`r = roots(c)` returns a column vector whose elements are the roots of the polynomial `c`.

Row vector `c` contains the coefficients of a polynomial, ordered in descending powers. If `c` has `n+1` components, the polynomial it represents is $c_1 s^n + \ldots + c_n s + c_{n+1}$.

Tips

Note the relationship of this function to `p = poly(r)`, which returns a row vector whose elements are the coefficients of the polynomial. For vectors, `roots` and `poly` are inverse functions of each other, up to ordering, scaling, and roundoff error.

Examples

The polynomial $s^3 - 6s^2 - 72s - 27$ is represented in MATLAB software as

```
p = [1 -6 -72 -27]
```

The roots of this polynomial are returned in a column vector by

```
r = roots(p)

r =
    12.1229
    -5.7345
    -0.3884
```

Algorithms

The algorithm simply involves computing the eigenvalues of the companion matrix:

```
A = diag(ones(n-1,1),-1);
A(1,:) = -c(2:n+1)./c(1);
eig(A)
```

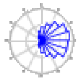
It is possible to prove that the results produced are the exact eigenvalues of a matrix within roundoff error of the companion matrix `A`, but this does not mean that they are the exact roots of a polynomial with coefficients within roundoff error of those in `c`.

See Also

`fzero` | `poly` | `residue`

Purpose

Angle histogram plot



Syntax

```
rose(theta)
rose(theta,x)
rose(theta,nbins)
rose(axes_handle,...)
h = rose(...)
[tout,rout] = rose(...)
```

Description

`rose(theta)` creates an angle histogram, which is a polar plot showing the distribution of values grouped according to their numeric range, showing the distribution of `theta` in 20 angle bins or less. The vector `theta`, expressed in radians, determines the angle of each bin from the origin. The length of each bin reflects the number of elements in `theta` that fall within a group, which ranges from 0 to the greatest number of elements deposited in any one bin.

`rose(theta,x)` uses the vector `x` to specify the number and the locations of bins. `length(x)` is the number of bins and the values of `x` specify the center angle of each bin. For example, if `x` is a five-element vector, `rose` distributes the elements of `theta` in five bins centered at the specified `x` values.

`rose(theta,nbins)` plots `nbins` equally spaced bins in the range `[0, 2*pi]`. The default is `20`.

`rose(axes_handle,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

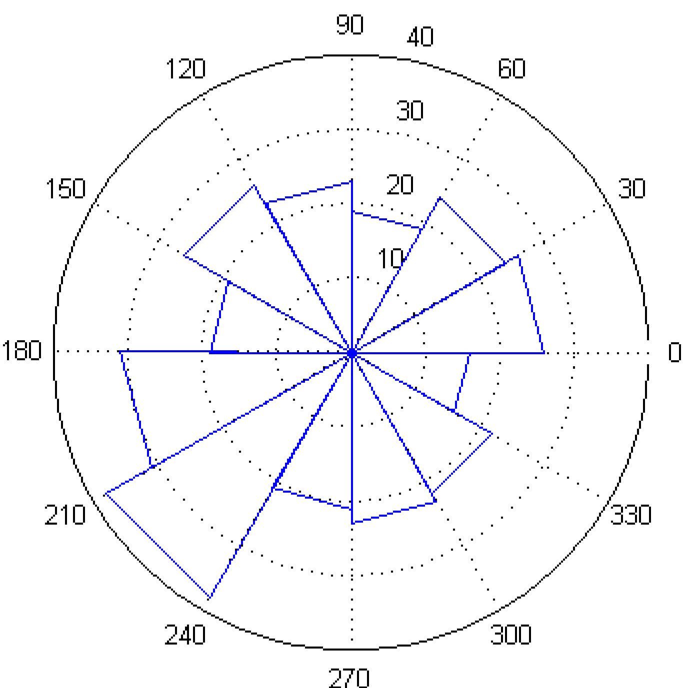
`h = rose(...)` returns the handles of the line objects used to create the graph.

`[tout,rout] = rose(...)` returns the vectors `tout` and `rout` so `polar(tout,rout)` generates the histogram for the data. This syntax does not generate a plot.

Examples

Load data on sunspot activity over 288 years. Generate a rose histogram of the number of sunspots observed using 12 bins.

```
figure
load sunspot.dat  % Contains a 2-column vector named sunspot
rose(sunspot(:,2),12)
```



See Also

`compass` | `feather` | `hist` | `line` | `polar`

How To

- Histograms in Polar Coordinates

Purpose

Classic symmetric eigenvalue test problem

Syntax

```
A = rosser
```

Description

`A = rosser` returns the Rosser matrix. The matrix is 8-by-8 with integer elements. It has:

- A double eigenvalue
- Three nearly equal eigenvalues
- Dominant eigenvalues of opposite sign
- A zero eigenvalue
- A small, nonzero eigenvalue

Examples

```
rosser

ans =

   611   196  -192   407    -8   -52   -49    29
   196   899   113  -192   -71   -43    -8   -44
  -192   113   899   196    61    49     8    52
   407  -192   196   611     8    44    59   -23
    -8   -71    61     8   411  -599   208   208
   -52   -43    49    44  -599   411   208   208
   -49    -8     8    59   208   208    99  -911
    29   -44    52   -23   208   208  -911    99
```

rot90

Purpose

Rotate matrix 90 degrees

Syntax

```
B = rot90(A)
B = rot90(A,k)
```

Description

`B = rot90(A)` rotates matrix `A` counterclockwise by 90 degrees.

`B = rot90(A,k)` rotates matrix `A` counterclockwise by `k*90` degrees, where `k` is an integer.

Examples

The matrix

```
X =
    1    2    3
    4    5    6
    7    8    9
```

rotated by 90 degrees is

```
Y = rot90(X)
Y =
    3    6    9
    2    5    8
    1    4    7
```

See Also

`flipdim` | `fliplr` | `flipud`

Purpose

Rotate object about specified origin and direction

Syntax

```
rotate(h,direction,alpha)
rotate(...,origin)
```

Description

The `rotate` function rotates a graphics object in three-dimensional space.

`rotate(h,direction,alpha)` rotates the graphics object `h` by `alpha` degrees. `direction` is a two- or three-element vector that describes the axis of rotation in conjunction with the origin of the axis of rotation. The default origin of the axis of rotation is the center of the plot box. This point is not necessarily the origin of the axes.

Positive `alpha` is defined as the righthand-rule angle about the direction vector as it extends from the origin of rotation.

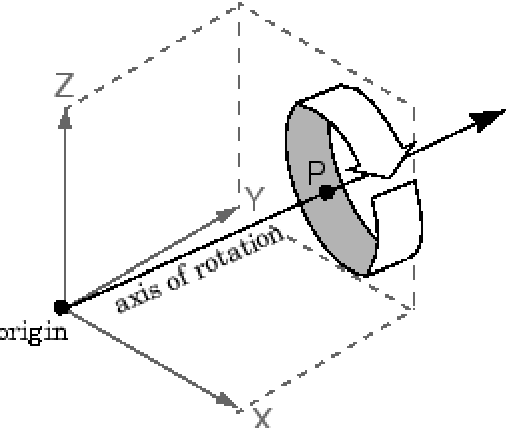
If `h` is an array of handles, all objects must be children of the same axes.

`rotate(...,origin)` specifies the origin of the axis of rotation as a three-element vector $[x_0,y_0,z_0]$.

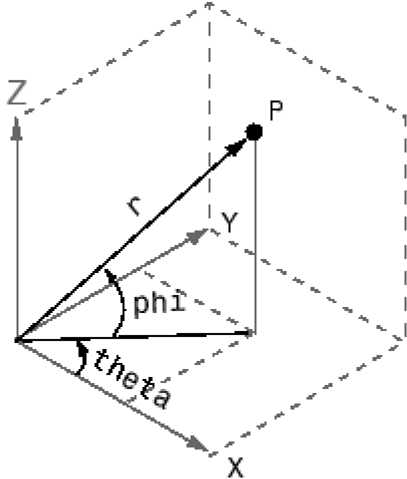
Tips

The rotation transformation modifies the object's data. This technique is different from that used by `view` and `rotate3d`, which modify only the viewpoint.

The axis of rotation is defined by an origin of rotation and a point `P`. Specify `P` as the spherical coordinates `[theta phi]` or as the Cartesian coordinates $[x_p,y_p,z_p]$.



In the two-element form for `direction`, `theta` is the angle in the *x*-*y* plane counterclockwise from the positive *x*-axis. `phi` is the elevation of the direction vector from the *x*-*y* plane.

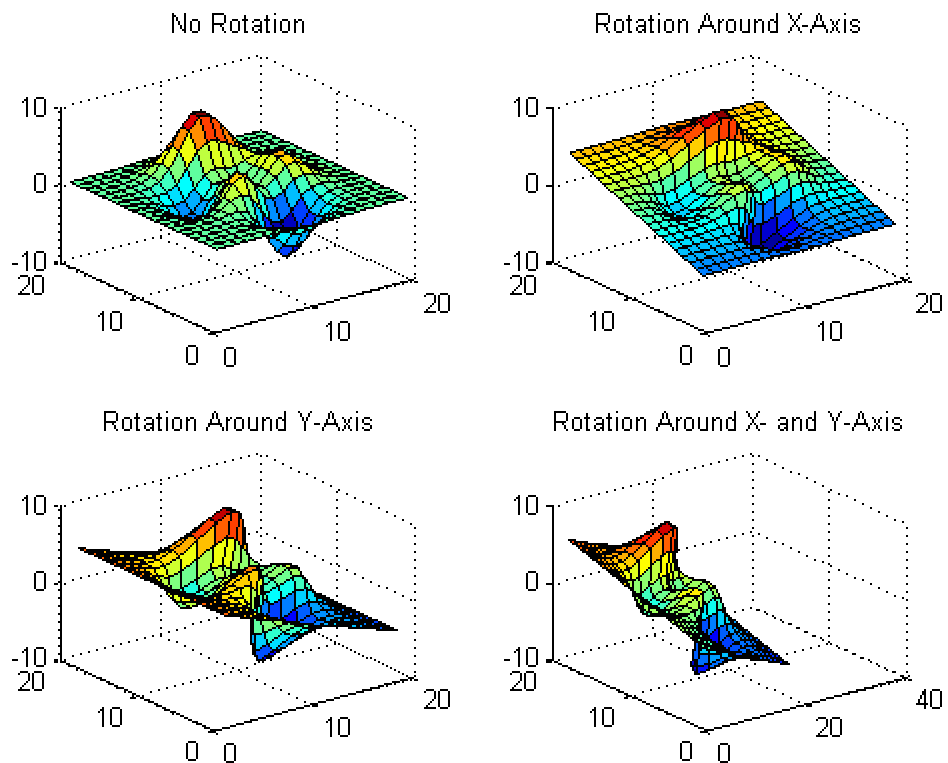


The three-element form for `direction` specifies the axis direction using Cartesian coordinates. The direction vector is the vector from the origin of rotation to `P`.

Examples

Create a 2-by-2 set of surface plots and rotate them around their *x* and *y* axes.

```
figure
sp11 = subplot(2,2,1);
h11 = surf(sp11, peaks(20));
title('No Rotation')
sp12 = subplot(2,2,2);
h12 = surf(sp12, peaks(20));
title('Rotation Around X-Axis')
zdir = [1 0 0];
rotate(h12,zdir,25)
sp21 = subplot(2,2,3);
h21 = surf(sp21, peaks(20));
title('Rotation Around Y-Axis')
zdir = [0 1 0];
rotate(h21,zdir,25)
sp22 = subplot(2,2,4);
h22 = surf(sp22, peaks(20));
title('Rotation Around X- and Y-Axis')
zdir = [1 1 0];
rotate(h22,zdir,25)
```



Tips

`rotate` changes the values of the `Xdata`, `Ydata`, and `Zdata` properties to rotate graphics objects.

See Also

`rotate3d` | `sph2cart` | `view` | `CameraPosition` | `CameraTarget` | `CameraUpVector` | `CameraViewAngle`

Purpose

Rotate 3-D view using mouse

Syntax

```
rotate3d on
rotate3d off
rotate3d
rotate3d(figure_handle,...)
rotate3d(axes_handle,...)
h = rotate3d(figure_handle)
```

Description

`rotate3d on` enables mouse-base rotation on all axes within the current figure.

`rotate3d off` disables interactive axes rotation in the current figure.

`rotate3d` toggles interactive axes rotation in the current figure.

`rotate3d(figure_handle,...)` enables rotation within the specified figure instead of the current figure.

`rotate3d(axes_handle,...)` enables rotation only in the specified axes.

`h = rotate3d(figure_handle)` returns a rotate3d *mode object* for figure figure_handle for you to customize the mode's behavior.

Using Rotate Mode Objects

You access the following properties of rotate mode objects via `get` and modify some of them using `set`.

- `FigureHandle <handle>` — The associated figure handle, a read-only property that cannot be set
- *`Enable`* `'on'|'off'` — Specifies whether this figure mode is currently enabled on the figure
- *`RotateStyle`* `'orbit'|'box'` — Sets the method of rotation

  `'orbit'` rotates the entire axes; `'box'` rotates a plot-box outline of the axes.

Rotate3D Mode Callbacks

You can program the following callbacks for rotate3d mode operations.

- ButtonDownFilter <function_handle> — Function to intercept ButtonDown events

  The application can inhibit the rotate operation under circumstances the programmer defines, depending on what the callback returns. The input function handle should reference a function with two implicit arguments (similar to handle callbacks):

  ```
  function [res] = myfunction(obj,event_obj)
  % obj          handle to the object that has been clicked on
  % event_obj    handle to event data object (empty in this release)
  % res [output] logical flag to determine whether the rotate
                 operation should take place or the 'ButtonDownFcn'
                 property of the object should take precedence
  ```

- ActionPreCallback <function_handle> — Function to execute before rotating

  Set this callback to listen to when a rotate operation will start. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks):

  ```
  function myfunction(obj,event_obj)
  % obj         handle to the figure that has been clicked on
  % event_obj   object containing struct of event data
  ```

  The event data has the following field:

Axes	The handle of the axes that is being panned

- ActionPostCallback <function_handle> — Function to execute after rotating

Set this callback to listen to when a rotate operation has finished. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks):

```
function myfunction(obj,event_obj)
% obj          handle to the figure that has been clicked on
% event_obj    object containing struct of event data (same as the
%              event data of the 'ActionPreCallback' callback)
```

Rotate3D Mode Utility Functions

The following functions in pan mode query and set certain of its properties.

- `flags = isAllowAxesRotate(h,axes)` — Function querying permission to rotate axes

  Calling the function `isAllowAxesRotate` on the rotate3d object, `h`, with a vector of axes handles, `axes`, as input will return a logical array of the same dimension as the axes handle vector which indicate whether a rotate operation is permitted on the axes objects.

- `setAllowAxesRotate(h,axes,flag)` — Function to set permission to pan axes

  Calling the function `setAllowAxesRotate` on the rotate3d object, `h`, with a vector of axes handles, `axes`, and a logical scalar, `flag`, will either allow or disallow a rotate operation on the axes objects.

Examples

Example 1

Simple 3-D rotation:

```
surf(peaks);
rotate3d on
% rotate the plot using the mouse pointer.
```

Example 2

Rotate the plot using the "Plot Box" rotate style:

```
surf(peaks);
h = rotate3d;
set(h,'RotateStyle','box','Enable','on');
% Rotate the plot.
```

Example 3

Create two axes as subplots and then prevent one from rotating:

```
ax1 = subplot(1,2,1);
surf(peaks);
h = rotate3d;
ax2 = subplot(1,2,2);
surf(membrane);
setAllowAxesRotate(h,ax2,false);
% rotate the plots.
```

Example 4

Create a buttonDown callback for rotate mode objects to trigger. Copy the following code to a new file, execute it, and observe rotation behavior:

```
function demo_mbd
% Allow a line to have its own 'ButtonDownFcn' callback.
hLine = plot(rand(1,10));
set(hLine,'ButtonDownFcn','disp(''This executes'')');
set(hLine,'Tag','DoNotIgnore');
h = rotate3d;
set(h,'ButtonDownFilter',@mycallback);
set(h,'Enable','on');
% mouse-click on the line
%
function [flag] = mycallback(obj,event_obj)
```

```
% If the tag of the object is 'DoNotIgnore', then return true.
objTag = get(obj,'Tag');
if strcmpi(objTag,'DoNotIgnore')
   flag = true;
else
   flag = false;
end
```

Example 5

Create callbacks for pre- and post-buttonDown events for rotate3D mode objects to trigger. Copy the following code to a new file, execute it, and observe rotation behavior:

```
function demo_mbd2
% Listen to rotate events
surf(peaks);
h = rotate3d;
set(h,'ActionPreCallback',@myprecallback);
set(h,'ActionPostCallback',@mypostcallback);
set(h,'Enable','on');
%
function myprecallback(obj,evd)
disp('A rotation is about to occur.');
%
function mypostcallback(obj,evd)
newView = round(get(evd.Axes,'View'));
msgbox(sprintf('The new view is [%d %d].',newView));
```

Tips

When enabled, `rotate3d` provides continuous rotation of axes and the objects it contains through mouse movement. A numeric readout appears in the lower left corner of the figure during rotation, showing the current azimuth and elevation of the axes. Releasing the mouse button removes the animated box and the readout. This differs from the `camorbit` function in that while the `rotate3d` tool modifies the `View` property of the axes, the `camorbit` function fixes the aspect ratio

and modifies the `CameraTarget`, `CameraPosition` and `CameraUpVector` properties of the axes. See `Axes Properties` for more information.

You can also enable 3-D rotation from the figure **Tools** menu or the figure toolbar.

You can create a `rotate3d` mode object once and use it to customize the behavior of different axes, as example 3 illustrates. You can also change its callback functions on the fly.

Note **Do not change figure callbacks within an interactive mode.** While a mode is active (when panning, zooming, etc.), you will receive a warning if you attempt to change any of the figure's callbacks and the operation will not succeed. The one exception to this rule is the figure `WindowButtonMotionFcn` callback, which can be changed from within a mode. Therefore, if you are creating a GUI that updates a figure's callbacks, the GUI should some keep track of which interactive mode is active, if any, before attempting to do this.

When you assign different 3-D rotation behaviors to different `subplot` axes via a mode object and then link them using the `linkaxes` function, the behavior of the axes you manipulate with the mouse will carry over to the linked axes, regardless of the behavior you previously set for the other axes.

Alternatives

Use the Rotate3D tool on the figure toolbar to enable and disable rotate3D mode on a plot, or select **Rotate 3D** from the figure's **Tools** menu. For details, see "Rotate in 3-D".

See Also

`camorbit` | `pan` | `rotate` | `view` | `zoom` | `Axes Properties`

Purpose

Round to nearest integer

Syntax

```
Y = round(X)
```

Description

`Y = round(X)` rounds the elements of `X` to the nearest integers. Positive elements with a fractional part of 0.5 round up to the nearest positive integer. Negative elements with a fractional part of -0.5 round down to the nearest negative integer. For complex `X`, the imaginary and real parts are rounded independently.

Examples

```
a = [-1.9, -0.2, 3.4, 5.6, 7.0, 2.4+3.6i]

a =
  Columns 1 through 4
  -1.9000            -0.2000             3.4000             5.6000
  Columns 5 through 6
   7.0000             2.4000 + 3.6000i

round(a)

ans =
  Columns 1 through 4
  -2.0000                  0             3.0000             6.0000
  Columns 5 through 6
   7.0000             2.0000 + 4.0000i
```

See Also

`ceil` | `fix` | `floor`

Purpose

Reduced row echelon form

Syntax

```
R = rref(A)
[R,jb] = rref(A)
[R,jb] = rref(A,tol)
```

Description

`R = rref(A)` produces the reduced row echelon form of `A` using Gauss Jordan elimination with partial pivoting. A default tolerance of (`max(size(A))*eps *norm(A,inf)`) tests for negligible column elements.

`[R,jb] = rref(A)` also returns a vector `jb` such that:

- `r = length(jb)` is this algorithm's idea of the rank of `A`.
- `x(jb)` are the pivot variables in a linear system `Ax = b`.
- `A(:,jb)` is a basis for the range of `A`.
- `R(1:r,jb)` is the `r`-by-`r` identity matrix.

`[R,jb] = rref(A,tol)` uses the given tolerance in the rank tests.

Roundoff errors may cause this algorithm to compute a different value for the rank than `rank`, `orth` and `null`.

Examples

Use `rref` on a rank-deficient magic square:

```
A = magic(4), R = rref(A)

A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

R =
     1     0     0     1
     0     1     0     3
     0     0     1    -3
```

```
     0     0     0     0
```

See Also

`inv` | `lu` | `rank`

rsf2csf

Purpose

Convert real Schur form to complex Schur form

Syntax

```
[U,T] = rsf2csf(U,T)
```

Description

The *complex Schur form* of a matrix is upper triangular with the eigenvalues of the matrix on the diagonal. The *real Schur form* has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal.

`[U,T] = rsf2csf(U,T)` converts the real Schur form to the complex form.

Arguments `U` and `T` represent the unitary and Schur forms of a matrix `A`, respectively, that satisfy the relationships: `A = U*T*U'` and `U'*U = eye(size(A))`. See `schur` for details.

Examples

Given matrix `A`,

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

with the eigenvalues

```
4.8121    1.9202 + 1.4742i    1.9202 + 1.4742i    1.3474
```

Generating the Schur form of `A` and converting to the complex Schur form

```
[u,t] = schur(A);
[U,T] = rsf2csf(u,t)
```

yields a triangular matrix `T` whose diagonal (underlined here for readability) consists of the eigenvalues of `A`.

```
U =
```

```
-0.4916       -0.2756 - 0.4411i    0.2133 + 0.5699i   -0.3428
-0.4980       -0.1012 + 0.2163i   -0.1046 + 0.2093i    0.8001
-0.6751        0.1842 + 0.3860i   -0.1867 - 0.3808i   -0.4260
-0.2337        0.2635 - 0.6481i    0.3134 - 0.5448i    0.2466

T =

4.8121        -0.9697 + 1.0778i   -0.5212 + 2.0051i   -1.0067
     0         1.9202 + 1.4742i    2.3355              0.1117 + 1.6547i
     0              0              1.9202 - 1.4742i    0.8002 + 0.2310i
     0              0                   0              1.3474
```

See Also

schur

Purpose Run MATLAB script

Syntax

```
run(scriptname)
```

Description

`run(scriptname)` runs the MATLAB script specified by `scriptname`.

Input Arguments

scriptname - Full or relative script path

string

Full or relative script path to a MATLAB script, specified as a string. `scriptname` can specify any file type that MATLAB can execute, such as MATLAB script files, Simulink models, or MEX-files.

Tips

- `run` executes scripts not currently on the MATLAB path. However, you should use `cd` or `addpath` to navigate to or to add the appropriate folder, making a script executable by entering its name alone.
- `scriptname` can access any variables in the current workspace.
- `run` changes to the folder that contains the script, executes it, and resets back to the original folder. If the script itself changes folders, then `run` does not revert to the original folder, unless `scriptname` changes to the folder in which this script resides.
- If `scriptname` corresponds to both a `.m` file and a P-file residing in the same folder, then `run` executes the P-file. This occurs even if you specify `scriptname` with a `.m` extension.

Examples

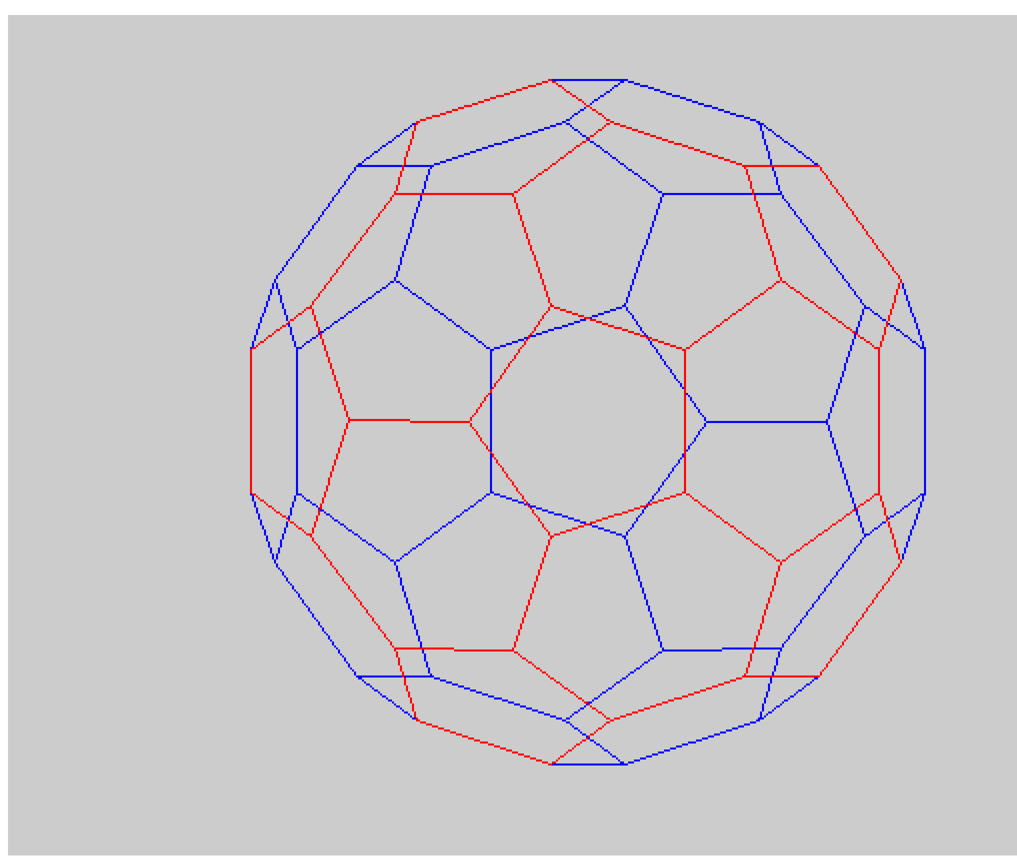
Run Script Not on Current Path

Create a temporary folder and copy an example MATLAB script to it.

```
tmp = tempname;
mkdir(tmp);
runtmp = fullfile(tmp,'buckyball.m');
demodir = fullfile(matlabroot,'toolbox','matlab','demos','buckydem.m');
copyfile(demodir,runtmp);
```

Run the new script.

```
run(runtmp)
```



See Also

`cd` | `path` | `addpath` | `pwd`

Concepts

- "Files and Folders that MATLAB Accesses"

Purpose

Save workspace variables to file

Syntax

```
save(filename)
save(filename, variables)
save(filename, '-struct', structName, fieldNames)
save(filename, ..., '-append')
save(filename, ..., format)
save(filename, ..., version)
save filename ...
```

Description

`save(filename)` stores all variables from the current workspace in a MATLAB formatted binary file (MAT-file) called `filename`.

`save(filename, variables)` stores only the specified variables.

`save(filename, '-struct', structName, fieldNames)` stores the fields of the specified scalar structure as individual variables in the file. If you include the optional `fieldNames`, the `save` function stores only the specified fields of the structure. You cannot specify `variables` and the `'-struct'` keyword in the same call to `save`.

`save(filename, ..., '-append')` adds new variables to an existing file. You can specify the `'-append'` option with additional inputs such as `variables`, `'-struct'` , `format`, or `version`.

`save(filename, ..., format)` saves in the specified format: `'-mat'` or `'-ascii'`. You can specify the `format` option with additional inputs such as `variables`, `'-struct'` , `'-append'`, or `version`.

`save(filename, ..., version)` saves to MAT-files in the specified version: `'-v4'`, `'-v6'`, `'-v7'`, or `'-v7.3'`. You can specify the `version` option with additional inputs such as `variables`, `'-struct'` , `'-append'`, or `format`.

`save filename ...` is the command form of the syntax, for convenient saving from the command line. With command syntax, you do not need to enclose input strings in single quotation marks. Separate inputs with spaces instead of commas. Do not use command syntax if inputs such as `filename` are variables. For more information, see “Command

vs. Function Syntax" in the MATLAB Programming Fundamentals documentation.

Input Arguments

filename

Name of a file. If you do not specify *filename*, the `save` function saves to a file named `matlab.mat`.

If *filename* does not include an extension and the value of *format* is `-mat` (the default), MATLAB appends `.mat`. If *filename* does not include a full path, MATLAB saves in the current folder. You must have permission to write to the file.

Default: `'matlab.mat'`

variables

Description of the variables to save. Use one of the following forms:

var1, var2, ...	Save the listed variables. Use the `'*'` wildcard to match patterns. For example, `save('A*')` saves all variables that start with `A`.
`'-regexp'`, *expressions*	Save only the variables that match the specified regular expressions. MATLAB treats all inputs as regular expressions, except the optional *filename*. The *filename* must appear immediately after the `save` command.

Default: all variables

'-struct'

Keyword to request saving the fields of a scalar structure as individual variables in the file. The *structName* input must appear immediately after the `-struct` keyword.

structName

Name of a scalar structure. Required when you use the `'-struct'` keyword.

fieldNames

Description of the fields of a structure to save as individual variables in the file. Use the same forms listed for *variables*. If you use the `'-regexp'` keyword, MATLAB treats all inputs as regular expressions except *filename* and *structName*.

'-append'

Keyword to add data to an existing file. For MAT-files, `-append` adds new variables to the file or replaces the saved values of existing variables with values in the workspace. For ASCII files, `-append` adds data to the end of the file.

format

Specifies the format of the file, regardless of any specified extension. Use one of the following combinations (not case sensitive):

`'-mat'`	Binary MAT-file format (default).
`'-ascii'`	8-digit ASCII format.
`'-ascii', '-tabs'`	Tab-delimited 8-digit ASCII format.
`'-ascii', '-double'`	16-digit ASCII format.
`'-ascii', '-double', '-tabs'`	Tab-delimited 16-digit ASCII format.

For MAT-files, data saved on one machine and loaded on another machine retains as much accuracy and range as the different machine floating-point formats allow.

For ASCII file formats, the `save` function has the following limitations:

- Each variable must be a two-dimensional `double` or character array.
- MATLAB translates characters to their corresponding internal ASCII codes. For example, `'abc'` appears in an ASCII file as:

```
9.7000000e+001   9.8000000e+001   9.9000000e+001
```

- The output includes only the real component of complex numbers.
- MATLAB writes data from each variable sequentially to the file. If you plan to use the `load` function to read the file, all variables must have the same number of columns. The `load` function creates a single variable from the file.

For more flexibility in creating ASCII files, use `dlmwrite` or `fprintf`.

version

Specifies the version of the file. Applies to MAT-files only.

The following table shows the available MAT-file version options and the corresponding supported features.

Option	Can Load in Versions	Supported Features
`'-v7.3'`	7.3 (R2006b) or later	Version 7.0 features plus support for data items greater than or equal to 2 GB on 64-bit systems.
`'-v7'`	7.0 (R14) or later	Version 6 features plus data compression and Unicode character encoding. Unicode encoding enables file sharing between systems that use different default character encoding schemes.

Option	Can Load in Versions	Supported Features
`'-v6'`	5 (R8) or later	Version 4 features plus *N*-dimensional arrays, cell arrays and structures, and variable names greater than 19 characters.
`'-v4'`	all	Two-dimensional `double`, character, and sparse arrays.

If any data items require features that the specified version does not support, MATLAB does not save those items and issues a warning. You cannot specify a version later than your version of MATLAB software.

To view or set the default version for MAT-files, select a **MAT-file save format** option in the General Preferences.

Examples

Save all variables from the workspace in binary MAT-file `test.mat`. Remove the variables from the workspace, and retrieve the data with the `load` function.

```
save test.mat
clear
load test.mat
```

Create a variable `savefile` that stores the name of a file, `pqfile.mat`. Save two variables to the file.

```
savefile = 'pqfile.mat';
p = rand(1, 10);
q = ones(10);
save(savefile, 'p', 'q')
```

Save data to an ASCII file, and view the contents of the file with the `type` function:

```
p = rand(1, 10);
q = ones(10);
save('pqfile.txt', 'p', 'q', '-ASCII')
type pqfile.txt
```

Alternatively, use command syntax for the `save` operation:

```
save pqfile.txt p q -ASCII
```

Save the fields of structure `s1` as individual variables. Check the contents of the file with the `whos` function. Clear the workspace and load the contents of a single field.

```
s1.a = 12.7;
s1.b = {'abc', [4 5; 6 7]};
s1.c = 'Hello!';

save('newstruct.mat', '-struct', 's1');

disp('Contents of newstruct.mat:')
whos('-file', 'newstruct.mat')

clear('s1')
load('newstruct.mat', 'b')
```

Save any variables in the workspace with names that begin with `Mon`, `Tue`, or `Wed` to `mydata.mat`:

```
save('mydata', '-regexp', '^Mon|^Tue|^Wed');
```

See Also

`clear` | `hgsave` | `load` | `matfile` | `regexp` | `saveas` | `whos` | `workspace`

How To

- "Save, Load, and Delete Workspace Variables"

- “Write to Delimited Data Files”
- “Supported File Formats”

Purpose

Serialize control object to file

Syntax

```
h.save('filename')
save(h, 'filename')
```

Description

`h.save('filename')` saves the COM control object, `h`, to the file specified in the string, `filename`.

`save(h, 'filename')` is an alternate syntax for the same operation.

Note The COM `save` function is only supported for controls at this time.

Tips

COM functions are available on Microsoft Windows systems only.

Examples

Create an `mwsamp` control and save its original state to the file `mwsample`:

```
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f);
h.save('mwsample')
```

Now, alter the figure by changing its label and the radius of the circle:

```
h.Label = 'Circle';
h.Radius = 50;
h.Redraw;
```

Using the `load` function, you can restore the control to its original state:

```
h.load('mwsample');
h.get
```

MATLAB displays the original values:

```
ans =
     Label: 'Label'
    Radius: 20
```

See Also

`load (COM)` | `actxcontrol` | `actxserver` | `release` | `delete (COM)`

Purpose

Save serial port objects and variables to file

Syntax

```
save filename
save filename obj1 obj2...
```

Description

`save filename` saves all MATLAB variables to the file `filename`. If an extension is not specified for `filename`, then the `.mat` extension is used.

`save filename obj1 obj2...` saves the serial port objects `obj1 obj2...` to the file `filename`.

Tips

You can use `save` in the functional form as well as the command form shown above. When using the functional form, you must specify the filename and serial port objects as strings. For example. to save the serial port object `s` to the file `MySerial.mat` on a Windows platform

```
s = serial('COM1');
save('MySerial','s')
```

Any data that is associated with the serial port object is not automatically stored in the file. For example, suppose there is data in the input buffer for `obj`. To save that data to a file, you must bring it into the MATLAB workspace using one of the synchronous read functions, and then save to the file using a separate variable name. You can also save data to a text file with the `record` function.

You return objects and variables to the MATLAB workspace with the `load` command. Values for read-only properties are restored to their default values upon loading. For example, the `Status` property is restored to `closed`. To determine if a property is read-only, examine its reference pages.

Examples

This example illustrates how to use the command and functional form of `save` on a Windows platform.

```
s = serial('COM1');
set(s,'BaudRate',2400,'StopBits',1)
save MySerial1 s
```

```
set(s,'BytesAvailableFcn',@mycallback)
save('MySerial2','s')
```

See Also

load | record | Status

Purpose

Save figure or Simulink block diagram using specified format

Alternatives

Use **File > Save As** on the figure window menu to access the Save As dialog, in which you can select a graphics format. For details, see "Exporting in a Specific Graphics Format" in the MATLAB Graphics documentation. Sizes of files written to image formats by this GUI and by `saveas` can differ due to disparate resolution settings.

Syntax

```
saveas(h,'filename.ext')
saveas(h,'filename','format')
```

Description

`saveas(h,'filename.ext')` saves the figure or Simulink block diagram with the handle `h` to the file `filename.ext`. The format of the file is determined by the extension, `ext`. Allowable values for `ext` are listed in this table.

You can pass the handle of any Handle Graphics object to `saveas`, which then saves the parent figure to the object you specified should `h` not be a figure handle. This means that `saveas` cannot save a subplot without also saving all subplots in its parent figure.

Text Value	Format
`ai`	Adobe Illustrator '88 Support for this format will be removed in a future release.
`bmp`	Windows bitmap
`emf`	Enhanced metafile
`eps`	EPS Level 1
`fig`	MATLAB figure (invalid for Simulink block diagrams)
`jpg`	JPEG image (invalid for Simulink block diagrams)
`m`	MATLAB file (invalid for Simulink block diagrams)

Text Value	Format
`pbm`	Portable bitmap
`pcx`	Paintbrush 24-bit
`pdf`	Portable Document Format
`pgm`	Portable Graymap
`png`	Portable Network Graphics
`ppm`	Portable Pixmap
`tif`	TIFF image, compressed

`saveas(h,'filename','format')` saves the figure or Simulink block diagram with the handle `h` to the file called `filename` using the specified `format`. The filename can have an extension, but the extension is not used to define the file format. If no extension is specified, the standard extension corresponding to the specified format is automatically appended to the filename.

Allowable values for `format` are the extensions in the table above and the device drivers and graphic formats supported by `print`. The drivers and graphic formats supported by `print` include additional file formats not listed in the table above. When using a `print` device type to specify format for `saveas`, do not prefix it with `-d`.

Tips

You can use `open` to open files saved using `saveas` with an `m` or `fig` extension. Other `saveas` and `print` formats are not supported by `open`. Both the **Save As** and **Export Setup** dialog boxes that you access from a figure's **File** menu use `saveas` with the `format` argument, and support all device and file types listed above.

Note Whenever you specify a format for saving a figure with the **Save As** menu item , that file format is used again the next time you save that figure or a new one. If you do not want to save in the previously-used format, use **Save As** and be sure to set the **Save as type** drop-down menu to the kind of file you want to write. However, saving a figure with the `saveas` function and a format does not change the **Save as type** setting in the GUI.

If you want to control the size or resolution of figures saved in image (bit-mapped) formats, such as `BMP` or `JPG`, use the `print` command and specify dots-per-inch resolution with the `r` switch.

Examples

Example 1: Specify File Extension

Save the current figure that you annotated using the Plot Editor to a file named `pred_prey` using the MATLAB `fig` format. This allows you to open the file `pred_prey.fig` at a later time and continue editing it with the Plot Editor.

```
saveas(gcf,'pred_prey.fig')
```

Example 2: Specify File Format but No Extension

Save the current figure, using a Portable Document format, to the file `logo`. Use the `pdf` extension from the above table to specify the format. The file created is `logo.pdf`.

```
saveas(gcf,'logo', 'pdf')
```

The file created is `logo.pdf`. MATLAB automatically appends the `pdf` extension because no extension was specified.

Example 3: Specify File Format and Extension

Save the current figure to the file `star.eps` using the Level 2 Color PostScript format. If you use `doc print` or `help print`, you can see from

the table for print device types that the device type for this format is `-dpsc2`. The file created is `star.eps`.

```
saveas(gcf,'star.eps', 'psc2')
```

In another example, save the current Simulink block diagram to the file `trans.tiff` using the TIFF format with no compression. From the table for print device types, you can see that the device type for this format is `-dtiffn`. The file created is `trans.tiff`.

```
saveas(gcf,'trans.tiff', 'tiffn')
```

Example 4: Saving a Simulink Diagram

Save a Simulink diagram from command line. The file is saved as `counters.bmp`

```
sldemo_tank
saveas(get_param('sldemo_tank','Handle'),'topmodel.bmp');
```

Using `get_param`, get the `handle` of the model and save using the `saveas` command. The file can be saved in any desired standard image formats.

See Also

`hgsave` | `open` | `print` | `save_system`

Purpose

Modify save process for object

Syntax

```
b = saveobj(a)
```

Description

`b = saveobj(a)` is called by the `save` function if the class of `a` defines a `saveobj` method. `save` writes the returned value, `b`, to the MAT-file.

Define a `loadobj` method to take the appropriate action when loading the object.

If `A` is an array of objects, MATLAB invokes `saveobj` separately for each object saved.

Examples

Call the superclass `saveobj` method from the subclass implementation of `saveobj` with the following syntax:

```
classdef mySub < super
   methods
      function sobj = saveobj(obj)
         % Call superclass saveobj method
         sobj = saveobj@super(obj);
         % Perform subclass save operations
         ...
      end
   ...
   end
...
end
```

See "Saving and Loading Objects from Class Hierarchies".

Update object when saved:

```
function b = saveobj(a)
   % If the object does not have an account number,
   % call method to add account number to AccountNumber property
   if isempty(a.AccountNumber)
      a.AccountNumber = getAccountNumber(a);
```

```
    end
    b = a;
end
```

See "Maintaining Class Compatibility".

See Also

`save` | `load` | `loadobj`

Tutorials

- "Control Save and Load"

Purpose

Save current search path

Alternatives

As an alternative to the `savepath` function, use the Set Path dialog box.

Syntax

```
savepath
savepath folderName/pathdef.m
status = savepath...
```

Description

`savepath` updates the MATLAB search path for all users on the system so that the path can be reused in a future session. `savepath` saves the search path to the `pathdef.m` file that MATLAB located at startup, or to the current folder if a `pathdef.m` file exists there. To save the search path programmatically each time you exit MATLAB, use `savepath` in a `finish.m` file.. On a Windows system with User Account Control (UAC) enabled, UAC might prompt you to allow the update operation because it requires administrator-level permission.

`savepath folderName/`**`pathdef.m`** saves the current search path to `pathdef.m` located in `folderName`. Use this form of the syntax if you do not have write access to the current `pathdef.m`. If you do not specify `folderName`, MATLAB saves `pathdef.m` in the current folder. `folderName` can be a relative or absolute path. To use the saved search path automatically in a future session, make `folderName` be the startup folder for MATLAB.

`status = savepath...` returns `0` when `savepath` was successful and `1` when `savepath` failed.

Examples

Save the current search path to `pathdef.m`, located in `I:/my_matlab_files`:

```
savepath I:/my_matlab_files/pathdef.m
```

See Also

`addpath` | `cd` | `dir` | `finish` | `genpath` | `matlabroot` | `pathsep` | `pathtool` | `rehash` | `restoredefaultpath` | `rmpath` | `startup` | `userpath` | `what`

How To

- "Running a Script When Exiting MATLAB"
- "What Is the MATLAB Search Path?"

Purpose

scatter plot

Syntax

```
scatter(X,Y)
scatter(X,Y,S)
scatter(X,Y,S,C)
scatter( ___ ,'fill')
scatter( ___ ,markertype)
scatter( ___ ,Name,Value)

scatter(axes_handle, ___ )

h = scatter( ___ )
```

Description

`scatter(X,Y)` displays circles at the locations specified by the vectors `X` and `Y`. This type of graph is also known as a bubble plot.

`scatter(X,Y,S)` draws each circle with the size specified by `S`. To plot each circle with equal size, specify `S` as a scalar. To plot each circle with a specific size, specify `S` as a vector with length equal to the length of `X` and `Y`.

`scatter(X,Y,S,C)` draws each circle with the color specified by `C`.

- If `C` is a color string or an RGB row vector, then all circles are plotted with the specified color.
- If `C` is a three column matrix with the number of rows in `C` equal to the length of `X` and `Y`, then each row of `C` specifies an RGB color value for the corresponding circle.
- If `C` is a vector with length equal to the length of `X` and `Y`, then the values in `C` are linearly mapped to the colors in the current colormap.

`scatter( ___ ,'fill')` fills in the circles, using any of the input argument combinations in the previous syntaxes.

`scatter(___,markertype)` specifies the marker type.

`scatter(___,Name,Value)` specifies scattergroup property settings using one or more `Name,Value` pair arguments.

`scatter(axes_handle,___)` plots into the axes specified by `axes_handle` instead of into the current axes (`gca`). The `axes_handle` option can precede any of the input argument combinations in the previous syntaxes.

`h = scatter(___)` returns the scattergroup object handle, `h`.

Input Arguments

X - Value of data to display on x-axis

vector

Value of data to display on the *x*-axis, specified as a vector. `X` and `Y` must be vectors of equal length.

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Y - Value of data to display on y-axis

vector

Value of data to display on the *y*-axis, specified as a vector. `X` and `Y` must be vectors of equal length.

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

S - Marker area

36 (default) | scalar | row or column vector | []

Marker area, specified as a scalar, a row or column vector, or `[]`. The values in `S` must be positive. The units for area are points squared.

- If S is a scalar, then scatter plots all markers with the specified area.
- If S is a row or column vector, then each entry in S specifies the area for the corresponding marker. The length of S must equal the length of X and Y, and corresponding entries in X, Y, and S determine the location and area of each marker.
- If S is empty, then the default size of 36 points squared is used.

Example: 50

Example: [36,25,25,17,46]

Data Types
single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

C - Marker color

[0 0 1] (blue) (default) | color string | RGB row vector | three-column matrix of RGB values | vector

Marker color, specified as a color string, an RGB row vector, a three-column matrix of RGB values, or a vector. For an RGB row vector, use a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0 1]. If you have three points in the scatter plot and want the colors to be indices into the colormap, specify C as a three-element column vector.

This table lists the predefined colors and their RGB equivalents.

RGB Vector	Short Name	Long Name
[1 1 0]	'y'	'yellow'
[1 0 1]	'm'	'magenta'
[0 1 1]	'c'	'cyan'
[1 0 0]	'r'	'red'
[0 1 0]	'g'	'green'

RGB Vector	Short Name	Long Name
[0 0 1]	'b'	'blue'
[1 1 1]	'w'	'white'
[0 0 0]	'k'	'black'

Example: `'k'`

Example: `[1,2,3,4]`

Example: `reshape([0,1,0,0,0,1,0.5,1,0.2],3,3)`

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

markertype - Marker type

`'o'` (default) | string

Marker type, specified as a string. This table lists the supported marker types.

Specifier	Marker Type
'o'	Circle
'+'	Plus sign
'*'	Asterisk
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle

Specifier	Marker Type
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No marker

Example: 'p'

Data Types
char

axes_handle - Axes handle

Axes handle, which is the reference to an axes object. Use the gca function to get the handle to the current axes, for example, axes_handle = gca;.

Name-Value Pair Arguments

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

Example: 'MarkerFaceColor','red' sets the marker face color to red.

For more information on these settings see scattergroup.

'LineWidth' - Line width

0.5 (default) | scalar

Line width, specified as the comma-separated pair consisting of 'LineWidth' and a scalar in points. The scalar sets the width size of the marker edge.

Example: `'LineWidth',0.75`

'MarkerEdgeColor' - Marker edge color

`[0 0 1]` (blue) (default) | `'auto'` | `'none'` | three-element RGB vector | string

Marker edge color, specified as the comma-separated pair consisting of `'MarkerEdgeColor'` and a color value. The color value can be one of the supported strings or an RGB vector, listed in the following tables.

Specifier	Result
'auto'	Uses same color as line color
'none'	Specifies no color, which makes unfilled markers invisible

For an RGB vector, use a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0 1]`. This table lists the predefined string colors and their RGB equivalents.

RGB Vector	Short Name	Long Name
[1 1 0]	'y'	'yellow'
[1 0 1]	'm'	'magenta'
[0 1 1]	'c'	'cyan'
[1 0 0]	'r'	'red'
[0 1 0]	'g'	'green'
[0 0 1]	'b'	'blue'
[1 1 1]	'w'	'white'
[0 0 0]	'k'	'black'

Example: `'MarkerEdgeColor',[1 .8 .1]`

'MarkerFaceColor' - Marker face color

'none' (default) | 'auto' | three-element RGB vector | string

Marker face color, specified as the comma-separated pair consisting of 'MarkerFaceColor' and a color value. MarkerFaceColor sets the fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). The color value can be one of the supported strings or an RGB vector, listed in the following tables.

Specifier	Result
'auto'	Uses same color as marker edge color
'none'	Makes the interior of the marker transparent, allowing the background to show through (default)

For an RGB vector, use a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0 1]. This table lists the predefined string colors and their RGB equivalents.

RGB Vector	Short Name	Long Name
[1 1 0]	'y'	'yellow'
[1 0 1]	'm'	'magenta'
[0 1 1]	'c'	'cyan'
[1 0 0]	'r'	'red'
[0 1 0]	'g'	'green'
[0 0 1]	'b'	'blue'
[1 1 1]	'w'	'white'
[0 0 0]	'k'	'black'

Example: 'MarkerFaceColor',[0 .8 1]

Output Arguments

h - Scattergroup object handle

scalar

Scattergroup object handle, returned as a scalar. This is a unique identifier, which you can use to query and modify the properties of the `scattergroup` object.

Examples

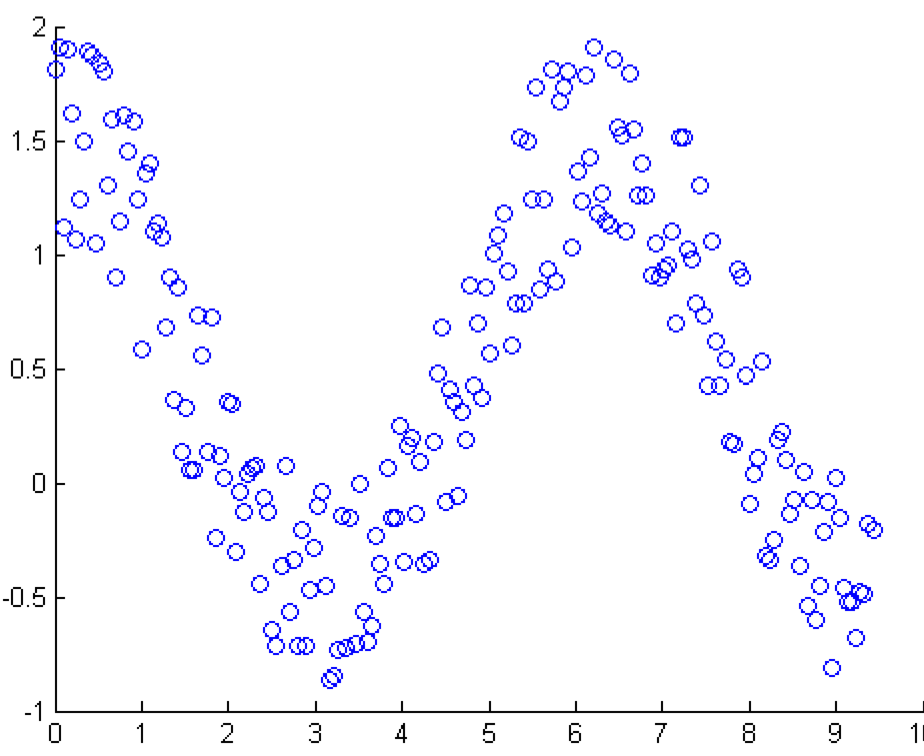
Create Scatter Plot

Set up vector `x` to contain equally spaced values between 0 and 3π. Initialize the random-number generator to make the output of `rand` repeatable, and set up vector `y` to contain cosine values with random noise.

```
x = linspace(0,3*pi,200);
rng(0,'twister');
y = cos(x)+ rand(1,200);
```

Create a scatter plot using the two vector inputs.

```
figure
scatter(x,y)
```



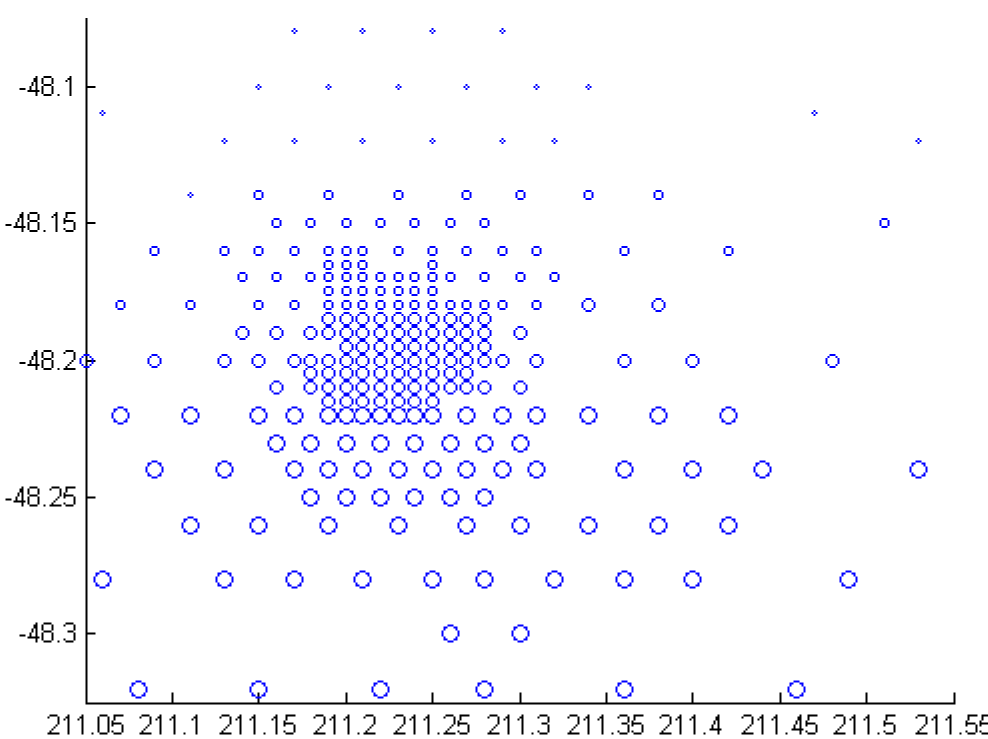
`scatter` plots entries in `x` against corresponding entries in `y`.

Vary Circle Size

Load sample data from `seamount` to get vectors `x` and `y`. Create a scatter plot and vary circle size. Use `zoom` to zoom in on the scatter plot.

```
figure
load seamount
S = linspace(1,50,length(x));
scatter(x,y,S)
```

```
zoom(2)
```



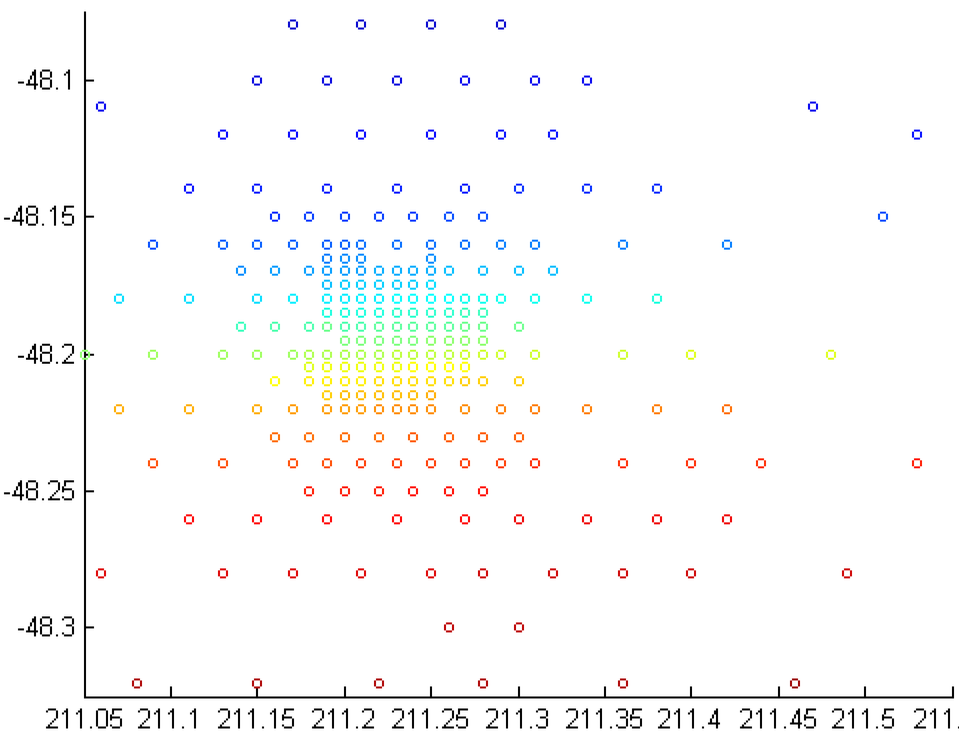
Corresponding entries in X, Y, and S determine the location and size of each marker.

Vary Circle Color

Load sample data from seamount to get vectors x and y. Create a scatter plot and vary the circle color. Use zoom to zoom in on the scatter plot.

```
figure
```

```
load seamount
s = 10;
c = linspace(1,10,length(x));
scatter(x,y,s,c)
zoom(2)
```

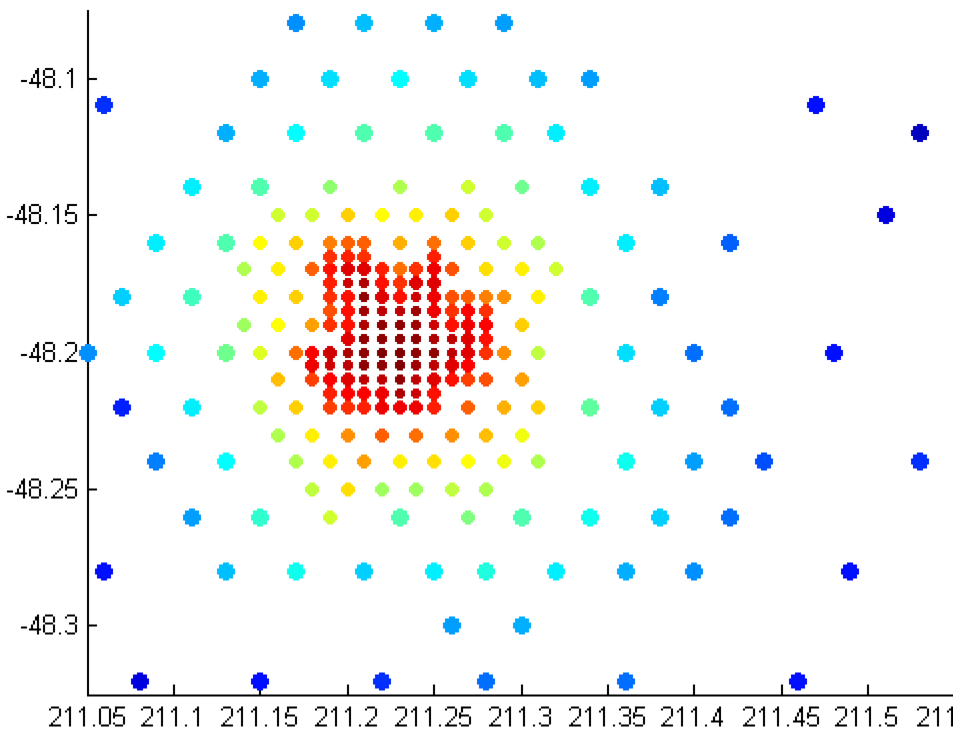


Corresponding entries in `X`, `Y`, and `C` determine the location and color of each marker. Since `S` is a scalar, all markers are plotted with equal size.

Vary Circle Size and Color

Load sample data from `seamount` to get vectors `x`, `y`, and `z`. Create a scatter plot and vary the circle size and color. Fill in the circles and use `zoom` to zoom in on the scatter plot.

```
figure
load seamount
s = sqrt(-z/2);
c = z;
scatter(x,y,s,c,'fill')
zoom(2)
```



Each circle is plotted with a specific size and color determined by the vectors `S` and `C`, respectively.

Specify Marker Symbol

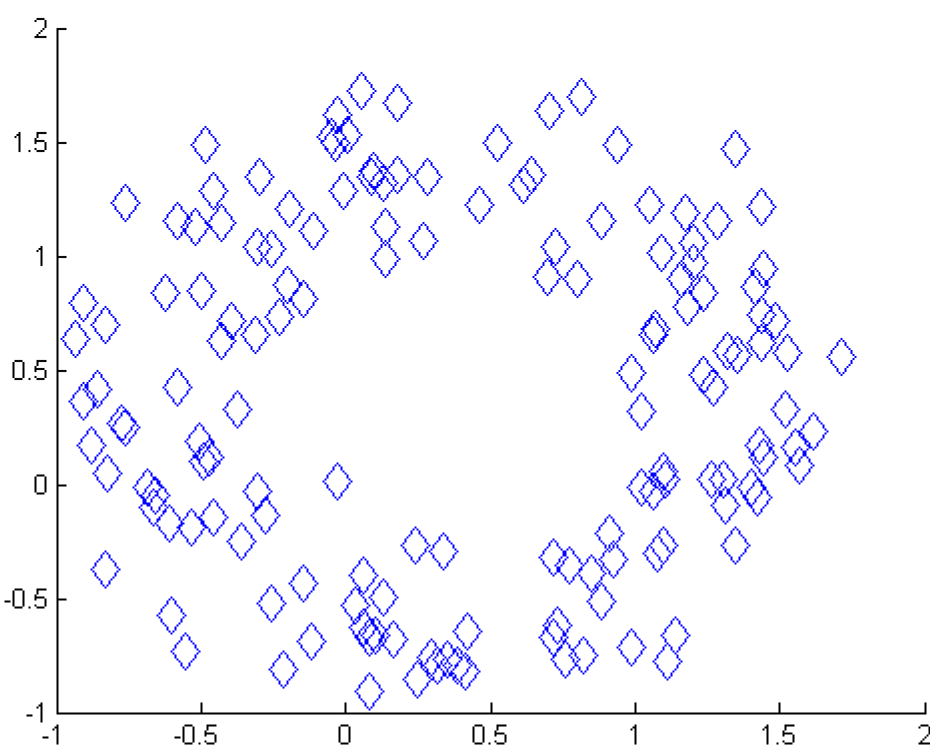
Initialize the random-number generator to make the output of `rand` repeatable. Set up vectors `x` and `y` as sine and cosine values with random noise.

```
rng(0,'twister');
theta = linspace(0,2*pi,150);
```

```
x = sin(theta) + 0.75*rand(1,150);
y = cos(theta) + 0.75*rand(1,150);
```

Create a scatter plot and set the marker type to diamonds with an area of 140 points squared.

```
figure
s = 140;
scatter(x,y,s,'d')
```



To fill in the diamonds, use the 'fill' option.

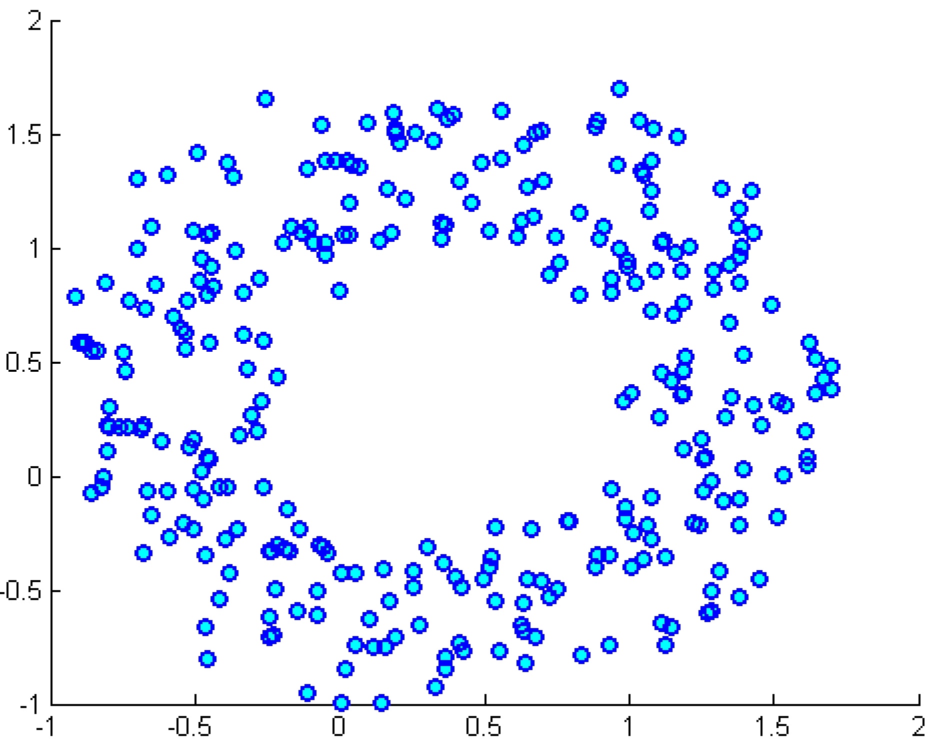
Specify Marker Properties

Initialize the random-number generator to make the output of rand repeatable. Set up vectors x and y as sine and cosine values with random noise.

```
rng(0,'twister');
theta = linspace(0,2*pi,300);
x = sin(theta) + 0.75*rand(1,300);
y = cos(theta) + 0.75*rand(1,300);
```

Create a scatter plot and set the marker size, edge color, face color, and line width.

```
figure
s = 40;
scatter(x,y,s,'MarkerEdgeColor','b','MarkerFaceColor','c','LineWidth',
```



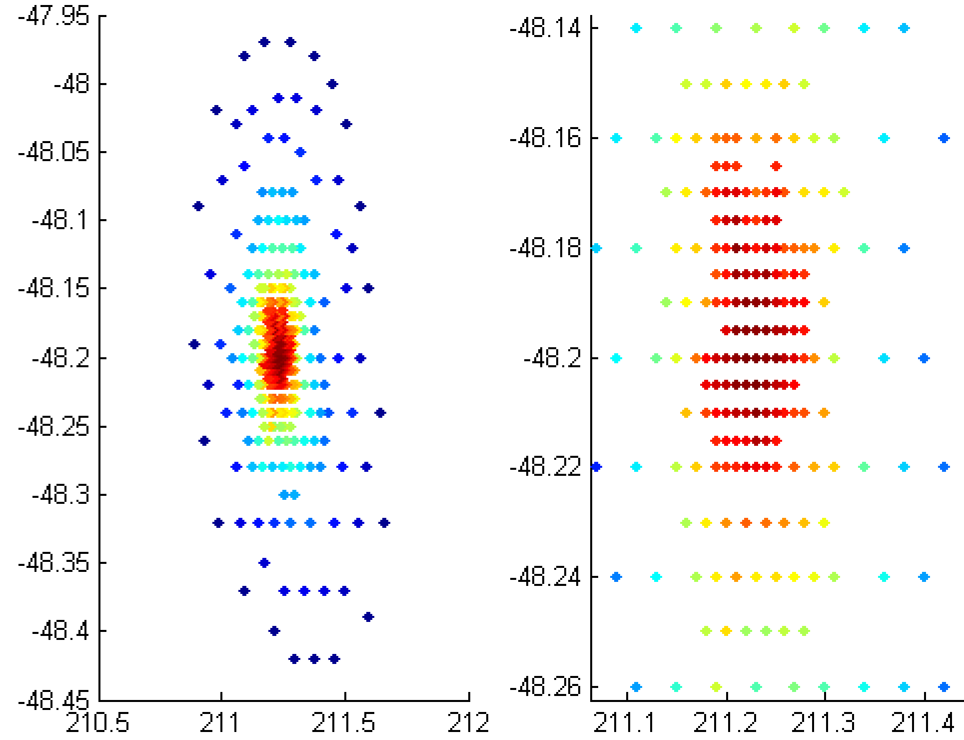
Specify Scatter Plot Axes

Load the sample data from `seamount` to get vectors `x`, `y`, and `z`.

```
load seamount
```

Create a figure with handles to two subplots. Create a scatter plot in the first subplot using the axes handle. In the second subplot, create another scatter plot from the same data sample and use `zoom` to zoom in on the plot.

```
figure
hs(1) = subplot(1,2,1);
hs(2) = subplot(1,2,2);
s = 30;
c = z;
scatter(hs(1),x,y,s,c,'fill','d')
scatter(hs(2),x,y,s,c,'fill','d')
zoom(4)
```



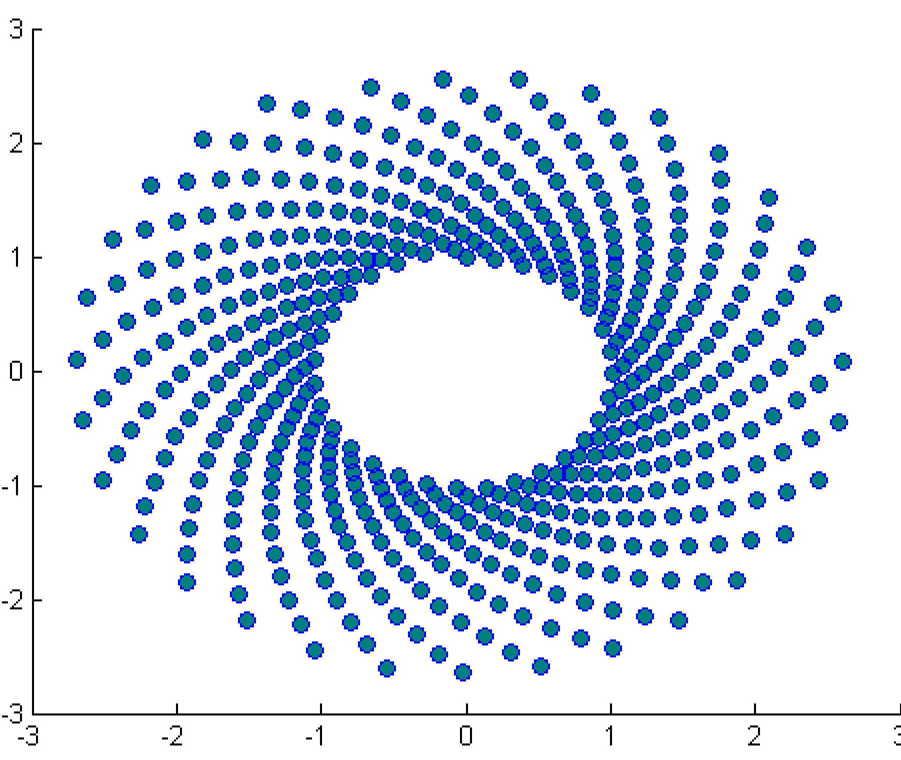
Set Marker Properties Using the Handle

Create a scatter plot and get the scattergroup object handle, h.

```
figure
theta = linspace(0,1,500);
x = exp(theta).*sin(100*theta);
y = exp(theta).*cos(100*theta);
h = scatter(x,y);
```

Use the handle to set the marker edge color, face color, and line width.

```
set(h,'MarkerEdgeColor','b','MarkerFaceColor',[0 .5 .5],'LineWidth',0.6)
```



See Also

scatter3 | plot3scattergroup | ColorSpec | LineSpec

scatter3

Purpose

3-D scatter plot

Syntax

```
scatter3(X,Y,Z)
scatter3(X,Y,Z,S)
scatter3(X,Y,Z,S,C)
scatter3( ___ ,'fill')
scatter3( ___ ,markertype)
scatter3( ___ ,Name,Value)

scatter3(axes_handle, ___ )

h = scatter3( ___ )
```

Description

`scatter3(X,Y,Z)` displays circles at the locations specified by the vectors `X`, `Y`, and `Z`.

`scatter3(X,Y,Z,S)` draws each circle with the size specified by `S`. To plot each circle with equal size, specify `S` as a scalar. To plot each circle with a specific size, specify `S` as a vector.

`scatter3(X,Y,Z,S,C)` draws each circle with the color specified by `C`.

- If `C` is a color string or an RGB row vector, then all circles are plotted with the specified color.
- If `C` is a three column matrix with the number of rows in `C` equal to the length of `X`, `Y`, and `Z`, then each row of `C` specifies an RGB color value for the corresponding circle.
- If `C` is a vector with length equal to the length of `X`, `Y`, and `Z`, then the values in `C` are linearly mapped to the colors in the current colormap.

`scatter3( ___ ,'fill')` fills in the circles, using any of the input argument combinations in the previous syntaxes.

`scatter3( ___ ,markertype)` specifies the marker type.

`scatter3(___,Name,Value)` specifies scattergroup property settings using one or more `Name,Value` pair arguments.

`scatter3(axes_handle,___)` plots into the axes specified by `axes_handle` instead of into the current axes (`gca`). The `axes_handle` option can precede any of the input argument combinations in the previous syntaxes.

`h = scatter3(___)` returns the scattergroup object handle, `h`.

Input Arguments

X - Value of data to display on x-axis

vector

Value of data to display on the *x*-axis, specified as a vector. `X`, `Y`, and `Z` must be vectors of equal length.

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Y - Value of data to display on y-axis

(default) | vector

Value of data to display on the *y*-axis, specified as a vector. `X`, `Y`, and `Z` must be vectors of equal length.

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Z - Value of data to display on z-axis

(default) | vector

Value of data to display on the *z*-axis, specified as a vector. `X`, `Y`, and `Z` must be vectors of equal length.

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

S - Marker area

36 (default) | scalar | vector | []

Marker area, specified as a scalar, a vector, or []. The values in S must be positive. The units for area are points squared.

- If S is a scalar, then scatter3 plots all markers with the specified area.
- If S is a row or column vector, then each entry in S specifies the area for the corresponding marker. The length of S must equal the length of X, Y and Z. Corresponding entries in X, Y, Z and S determine the location and area of each marker.
- If S is empty, then the default size of 36 points squared is used.

Example: 50

Example: [36,25,25,17,46]

Data Types

single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

C - Marker color

[0 0 1] (blue) (default) | color string | RGB row vector | three-column matrix of RGB values | vector

Marker color, specified as a color string, an RGB row vector, a three-column matrix of RGB values, or a vector. For an RGB row vector, use a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0 1]. If you have three points in the scatter plot and want the colors to be indices into the colormap, specify C as a three-element column vector.

This table lists the predefined colors and their RGB equivalents.

RGB Vector	Short Name	Long Name
[1 1 0]	'y'	'yellow'
[1 0 1]	'm'	'magenta'
[0 1 1]	'c'	'cyan'
[1 0 0]	'r'	'red'
[0 1 0]	'g'	'green'
[0 0 1]	'b'	'blue'
[1 1 1]	'w'	'white'
[0 0 0]	'k'	'black'

Example: 'y'

Example: [1,2,3,4]

Example: reshape([0,1,0,0,0,1,0.5,1,0.2],3,3)

Data Types
single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

markertype - Marker type

'o' (default) | string

Marker type, specified as a string. The table below lists the supported marker types.

Specifier	Marker Type
'o'	Circle
'+'	Plus sign
'*'	Asterisk
'.'	Point
'x'	Cross

Specifier	Marker Type
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No marker

axes_handle - Axes handle

Axes handle, which is the reference to an axes object. Use the `gca` function to get the handle to the current axes, for example, `axes_handle = gca;`.

Name-Value Pair Arguments

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

Example: `'MarkerFaceColor','red'` sets the marker face color to red.

For more information on these properties see `scattergroup`.

'LineWidth' - Line width

`0.5` (default) | scalar

Line width, specified as the comma-separated pair consisting of 'LineWidth' and a scalar. The scalar sets the width size in points of the marker edge.

Example: 'LineWidth',0.75

'MarkerEdgeColor' - Marker edge color

[0 0 1] (blue) (default) | 'auto' | 'none' | three-element RGB vector | string

Marker edge color, specified as the comma-separated pair consisting of 'MarkerEdgeColor' and a color value. The color value can be one of the supported strings or an RGB vector, listed in the following tables.

Specifier	Result
'auto'	Uses same color as line color
'none'	Specifies no color, which makes unfilled markers invisible

For an RGB vector, use a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0 1]. This table lists the predefined string colors and their RGB equivalents.

RGB Vector	Short Name	Long Name
[1 1 0]	'y'	'yellow'
[1 0 1]	'm'	'magenta'
[0 1 1]	'c'	'cyan'
[1 0 0]	'r'	'red'
[0 1 0]	'g'	'green'
[0 0 1]	'b'	'blue'
[1 1 1]	'w'	'white'
[0 0 0]	'k'	'black'

Example: `'MarkerEdgeColor',[1 .8 .1]`

'MarkerFaceColor' - Marker face color

`'none'` (default) | `'auto'` | three-element RGB vector | string

Marker face color, specified as the comma-separated pair consisting of `'MarkerFaceColor'` and a color value. `MarkerFaceColor` sets the fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). The color value can be one of the supported strings or an RGB vector, listed in the following tables.

Specifier	Result
'auto'	Uses same color as marker edge color
'none'	Makes the interior of the marker transparent, allowing the background to show through (default)

For an RGB vector, use a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0 1]`. This table lists the predefined string colors and their RGB equivalents.

RGB Vector	Short Name	Long Name
[1 1 0]	'y'	'yellow'
[1 0 1]	'm'	'magenta'
[0 1 1]	'c'	'cyan'
[1 0 0]	'r'	'red'
[0 1 0]	'g'	'green'
[0 0 1]	'b'	'blue'
[1 1 1]	'w'	'white'
[0 0 0]	'k'	'black'

Example: `'MarkerFaceColor',[0 .8 1]`

Output Arguments

h - Scattergroup object handle

scalar

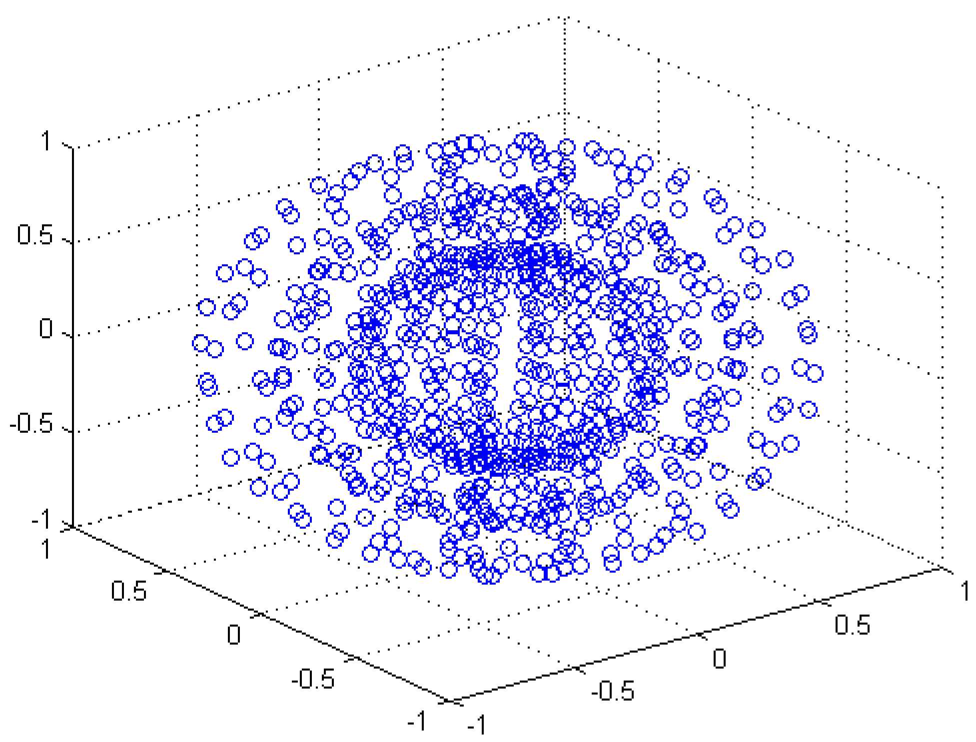
Scattergroup object handle, returned as a scalar. This is a unique identifier, which you can use to query and modify the properties of the `scattergroup`.

Examples

Create 3-D Scatter Plot

Create a 3-D scatter plot. Use `sphere` to set up vectors `x`, `y`, and `z`.

```
figure
[X,Y,Z] = sphere(16);
x = [0.5*X(:); 0.75*X(:); X(:)];
y = [0.5*Y(:); 0.75*Y(:); Y(:)];
z = [0.5*Z(:); 0.75*Z(:); Z(:)];
scatter3(x,y,z)
```



Vary Marker Size

Use `sphere` to set up vectors `x`, `y`, and `z`.

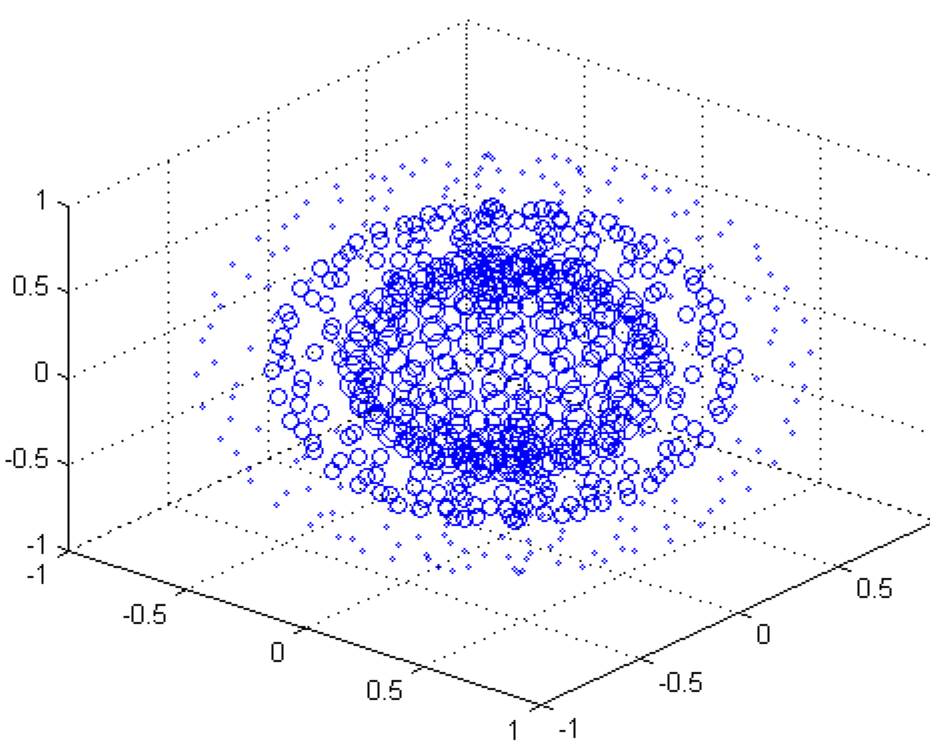
```
[X,Y,Z] = sphere(16);
x = [0.5*X(:); 0.75*X(:); X(:)];
y = [0.5*Y(:); 0.75*Y(:); Y(:)];
z = [0.5*Z(:); 0.75*Z(:); Z(:)];
```

Set up vector `s` to specify the marker sizes.

```
S = repmat([100,50,5],numel(X),1);
s = S(:);
```

Create a 3-D scatter plot and use `view` to change the angle of the axes in the figure.

```
figure
scatter3(x,y,z,s)
view(40,35)
```



Corresponding entries in x, y, z, and s determine the location and size of each marker.

Vary Marker Color

Use sphere to set up vectors x, y, and z.

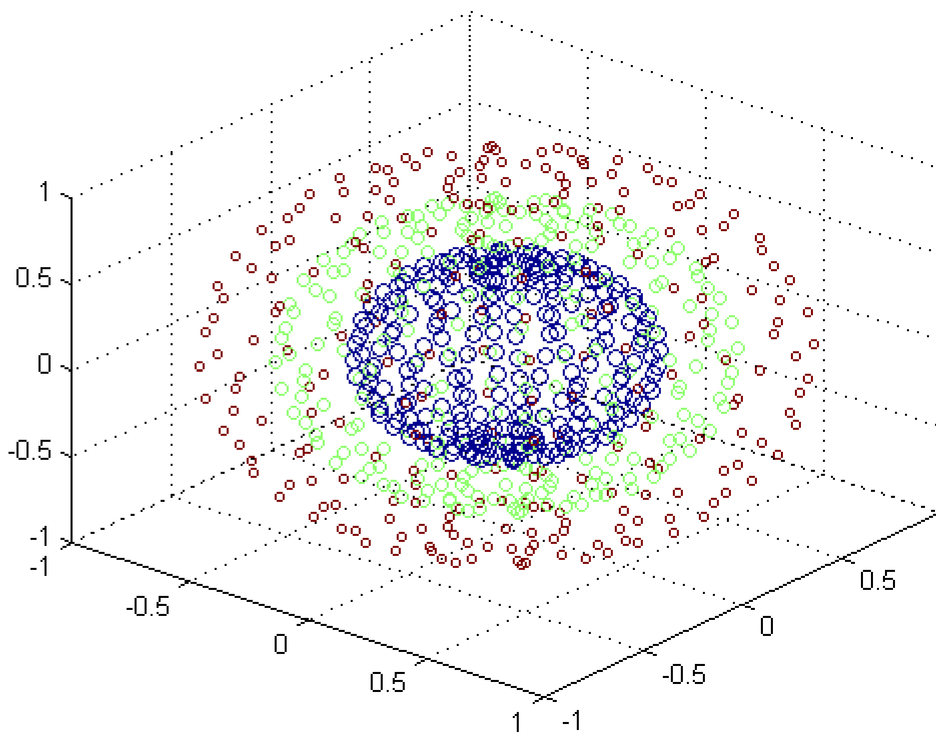
```
[X,Y,Z] = sphere(16);
x = [0.5*X(:); 0.75*X(:); X(:)];
y = [0.5*Y(:); 0.75*Y(:); Y(:)];
z = [0.5*Z(:); 0.75*Z(:); Z(:)];
```

Set up vectors s and c to specify the size and color of each marker.

```
S = repmat([50,25,10],numel(X),1);
C = repmat([1,2,3],numel(X),1);
s = S(:);
c = C(:);
```

Create a 3-D scatter plot and use view to change the angle of the axes in the figure.

```
figure
scatter3(x,y,z,s,c)
view(40,35)
```



Corresponding entries in x, y, z, and c determine the location and color of each marker.

Fill in Markers

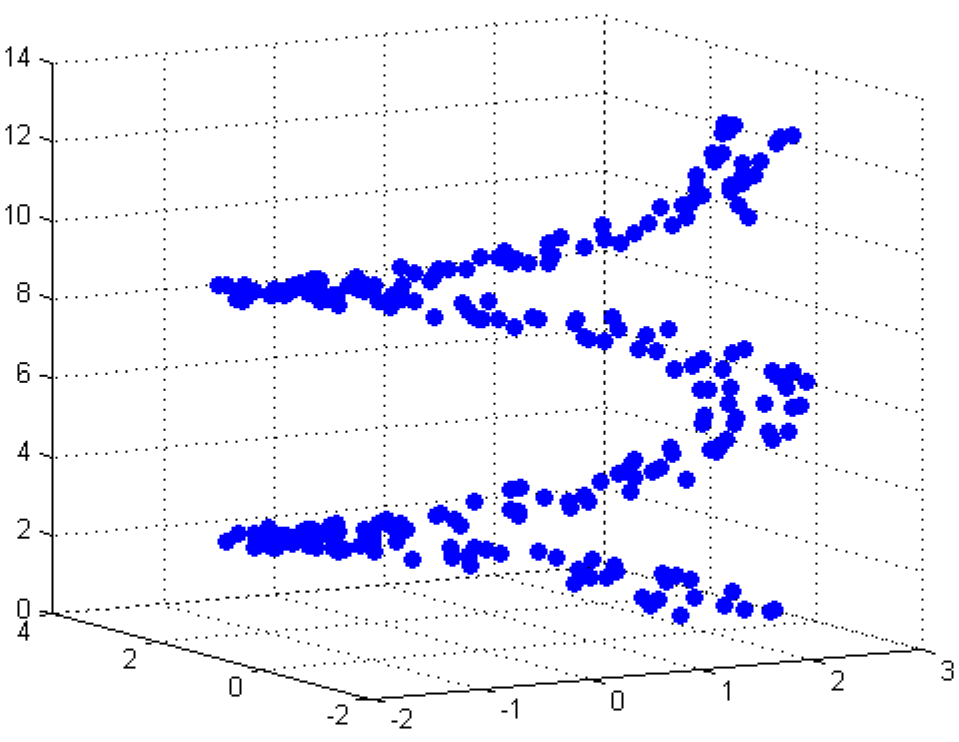
Initialize the random-number generator to make the output of rand repeatable. Set up vectors x and y as cosine and sine values with random noise.

```
rng(0,'twister');
z = linspace(0,4*pi,250);
```

```
x = 2*cos(z) + rand(1,250);
y = 2*sin(z) + rand(1,250);
```

Create a 3-D scatter plot and fill in the markers. Use `view` to change the angle of the axes in the figure.

```
figure
scatter3(x,y,z,'fill')
view(-30,10)
```



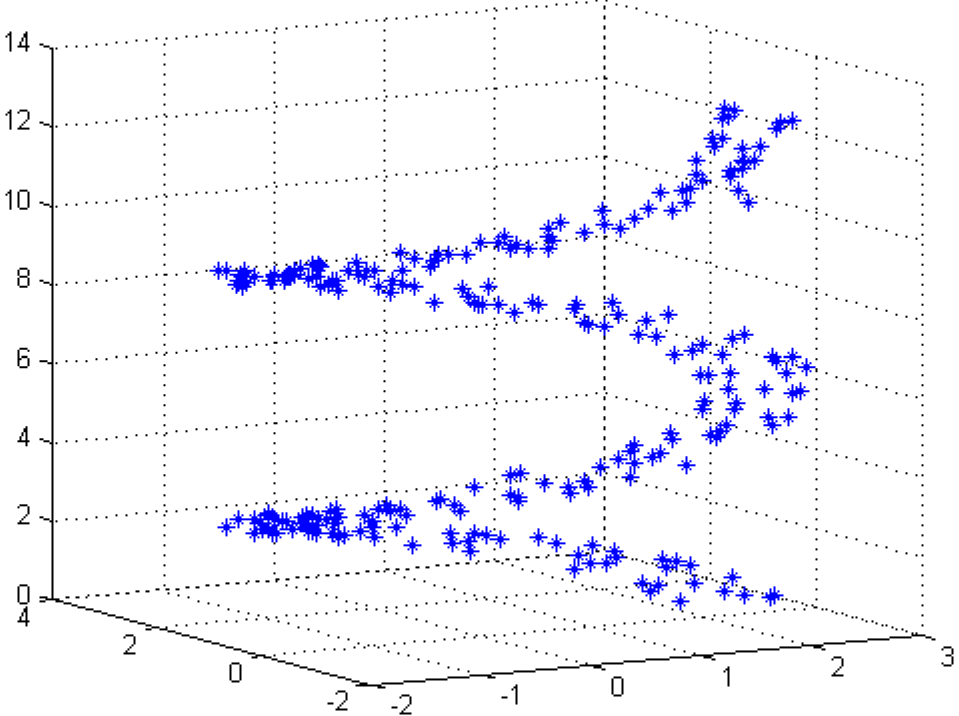
Set Marker Type

Initialize the random-number generator to make the output of `rand` repeatable. Set up vectors `x` and `y` as cosine and sine values with random noise.

```
rng(0,'twister');
z = linspace(0,4*pi,250);
x = 2*cos(z) + rand(1,250);
y = 2*sin(z) + rand(1,250);
```

Create a 3-D scatter plot and set the marker type. Use `view` to change the angle of the axes in the figure.

```
figure
scatter3(x,y,z,'*')
view(-30,10)
```



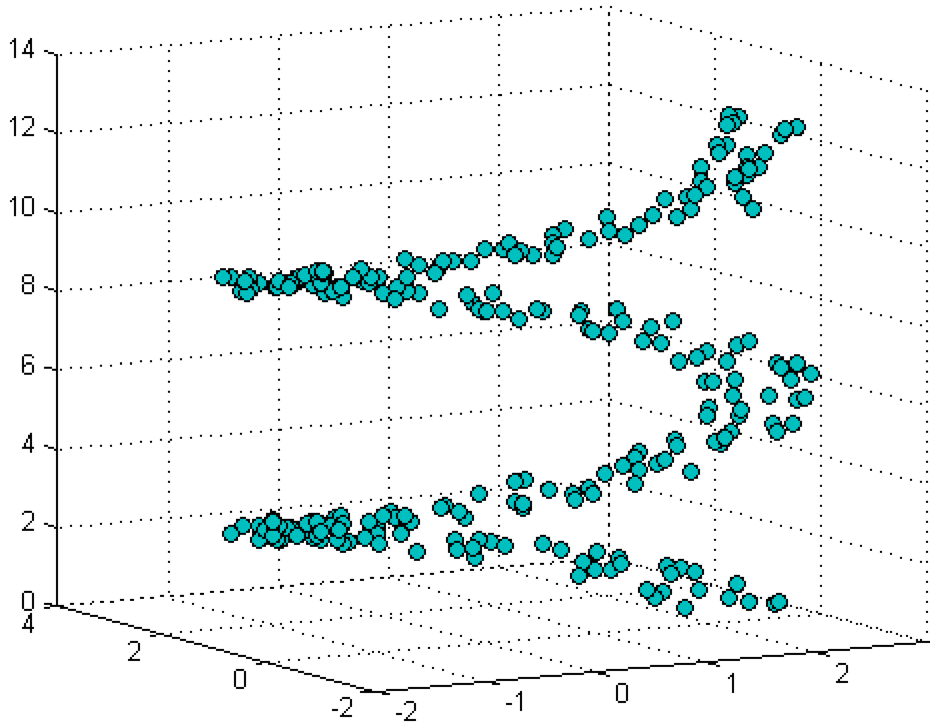
Set Marker Properties

Initialize the random-number generator to make the output of `rand` repeatable. Set up vectors `x` and `y` as cosine and sine values with random noise.

```
rng(0,'twister');
z = linspace(0,4*pi,250);
x = 2*cos(z) + rand(1,250);
y = 2*sin(z) + rand(1,250);
```

Create a 3-D scatter plot and set the marker edge color and the marker face color. Use `view` to change the angle of the axes in the figure.

```
figure
scatter3(x,y,z,'MarkerEdgeColor','k','MarkerFaceColor',[0 .75 .75])
view(-30,10)
```



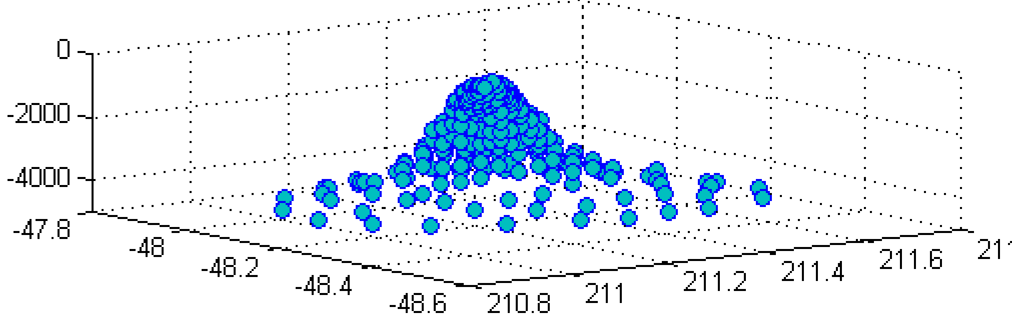
Specify 3-D Scatter Plot Axes

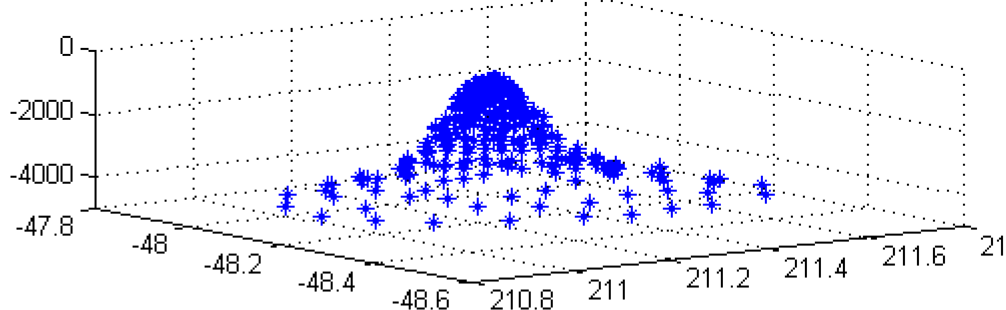
Load the `seamount` data set to get vectors `x`, `y`, and `z`.

```
load seamount
```

Create a figure with handles to two subplots. In each subplot create a 3-D scatter plot and specify the marker properties for each scatter plot.

```
figure
hs(1) = subplot(2,1,1);
hs(2) = subplot(2,1,2);
scatter3(hs(1),x,y,z,'MarkerFaceColor',[0 .75 .75])
scatter3(hs(2),x,y,z,'*')
```





Set Marker Properties Using the Handle

Use `sphere` to set up vectors `x`, `y`, and `z`.

```
[X,Y,Z] = sphere(16);
x = [0.5*X(:); 0.75*X(:); X(:)];
y = [0.5*Y(:); 0.75*Y(:); Y(:)];
z = [0.5*Z(:); 0.75*Z(:); Z(:)];
```

Set up vectors `s` and `c` to specify the size and color for each marker.

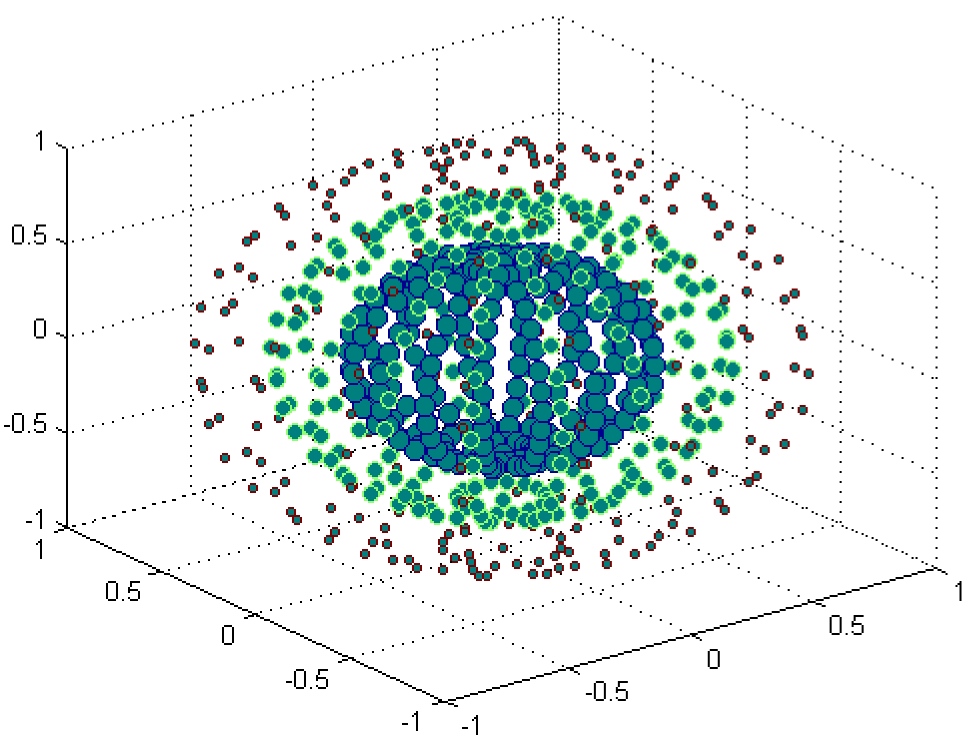
```
S = repmat([70,50,20],numel(X),1);
C = repmat([1,2,3],numel(X),1);
s = S(:);
c = C(:);
```

Create a 3-D scatter plot and get the object handle.

```
figure
h = scatter3(x,y,z,s,c);
```

Use the object handle to set the marker face color.

```
set(h,'MarkerFaceColor',[0 .5 .5])
```



See Also scatter | plot3scattergroup | | ColorSpec | LineSpec

Purpose

Define scattergroup properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands or the Property Editor (`propertyeditor`).

Note that you cannot define default property values for scattergroup objects.

See Plot Objects for information on scattergroup objects.

Scattergroup Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`Annotation`

`hg.Annotation` object (read-only)

Control the display of scattergroup objects in legends. Specifies whether this scattergroup object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the scattergroup object is displayed in a figure legend.

IconDisplayStyle Value	Purpose
`on`	Include the scattergroup object in a legend as one entry, but not its children objects
`off`	Do not include the scattergroup or its children in a legend (default)
`children`	Include only the children of the scattergroup as separate entries in the legend

Setting the IconDisplayStyle Property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the IconDisplayStyle Property

See "Controlling Legends" for more information and examples.

`BeingDeleted`

`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the

running callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`

string | function handle

Button press callback function. Executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be:

- A string that is a valid MATLAB expression
- The name of a MATLAB file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`CData`

vector | `m-by-3` matrix | `ColorSpec`

Color of markers. When `CData` is a vector the same length as `XData` and `YData`, the values in `CData` are linearly mapped to the colors in the current colormap. When `CData` is a `length(XData)`-by-3 matrix, it specifies the colors of the markers as RGB values.

`CDataSource`

string (MATLAB variable)

Link CData to MATLAB variable. Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the `CData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `CData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`. See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`Children`

array of graphics object handles

Children of the scattergroup object. An array containing the handle of a patch object parented to the scattergroup object (whether visible or not).

If a child object's `HandleVisibility` property is `callback` or `off`, its handle does not show up in this object's `Children` property. If you want the handle in the `Children` property, set the root `ShowHiddenHandles` property to `on`. For example:

```
set(0,'ShowHiddenHandles','on')
```

`Clipping`

`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs appear outside the axes plot box. This occurs if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`CreateFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback function executed during object creation. Defines a callback that executes when MATLAB creates an object. The default is an empty array.

You must specify the callback during the creation of the object. For example:

```
graphicfcn(y,'CreateFcn',@CallbackFcn)
```

where *`@CallbackFcn`* is a function handle that references the callback function and *`graphicfcn`* is the plotting function which creates this object.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DeleteFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback executed during object deletion. Executes when this object is deleted (for example, this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values. The default is an empty array.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`

string

String used by legend. The `legend` function uses the `DisplayName` property to label the scattergroup object in the legend. The default is an empty string.

- If you specify string arguments with the `legend` function, MATLAB set `DisplayName` to the corresponding string and uses that string for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, MATLAB set `DisplayName` to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add a legend programmatically that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more information and examples.

`EraseMode`

`{normal} | none | xor | background`

Erase mode. Controls the technique MATLAB uses to draw and erase objects. Use alternative erase modes for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the object when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it because MATLAB stores no information about its former location.

- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object's color depends on the color of whatever is beneath it on the display.
- `background` — Erase the object by redrawing it in the axes background `Color`, or the figure background `Color` if the axes `Color` property is `none`. This damages objects that are behind the erased object, but properly colors the erased object.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors (for example, performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible.

- `callback` — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties. See also `findall`.

Handle Validity

Hidden handles are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`
`{on} | off`

Selectable by mouse click. Determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the markers that compose the scatter plot. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`HitTestArea`
`on | {off}`

Select the object by clicking markers or area of extent. Select plot objects by:

- Clicking scatter markers (default).
- Clicking anywhere in the extent of the plot.

When `HitTestArea` is `off`, you must click the scatter markers to select the object. When `HitTestArea` is `on`, you can select this object by clicking anywhere within the extent of the plot (that is, anywhere within a rectangle that encloses all the scatter markers).

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the `Interruptible` property affects only the callbacks for the`ButtonDownFcn` property. A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both the callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see “Control Callback Execution and Interruption”.

`LineWidth`
size in points

Width of linear objects and edges of filled areas. Specify in points. 1 point = $^1/_{72}$ inch. The default is `0.5` points.

`Marker`
character (see table)

Marker symbol. Specifies marks that display at data points. You can set values for the `Marker` property independently from the `LineStyle` property. For a list of supported marker symbols, see the following table.

Marker Specifiers Table

Specifier	Marker Type
'`+`'	Plus sign
'`o`'	Circle
'`*`'	Asterisk
'`.`'	Point
'`x`'	Cross
'`square`' or '`s`'	Square

Specifier	Marker Type
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'''	Six-pointed star (hexagram)
'none'	No marker (default)

MarkerEdgeColor
ColorSpec | none | {auto}

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- ColorSpec — Defines color to use.
- none — Specifies no color, which makes nonfilled markers invisible.
- auto — Uses same color as the CData property.

MarkerFaceColor
ColorSpec | {none} | auto

Fill color for closed-shape markers. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- ColorSpec — User-defined color.
- none — Makes the interior of the marker transparent, allowing the background to show through.

- `auto` — Sets the fill color to the axes `Color` property. If the axes `Color` property is `none`, sets the fill color to the figure `Color`.

`Parent`
: handle of parent axes, hggroup, or hgtransform

  Parent of object. Handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

  See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
: `on | {off}`

  Object selection state. When you set this property to `on`, MATLAB displays selection handles at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
: `{on} | off`

  Object highlighted when selected.

  - `on` — MATLAB indicates the selected state by drawing four edge handles and four corner handles.
  - `off` — MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`SizeData`
: square points

  Size of markers in square points. Area of the marker in the scatter graph in units of points. Since there are 72 points to one inch, to

specify a marker that has an area of one square inch you would use a value of `72^2`.

`SizeDataSource`

string (MATLAB variable)

Link SizeData to MATLAB variable. Set this property to a MATLAB variable that, by default, is evaluated in the base workspace to generate the `SizeData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `SizeData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`. See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`Tag`

string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

For example, create an areaseries object and set the `Tag` property.

```
t = area(Y,'Tag','area1')
```

When you want to access objects of a given type, use `findobj` to find the object's handle. The following statement changes the `FaceColor` property of the object whose `Tag` is `area1`.

```
set(findobj('Tag','area1'),'FaceColor','red')
```

`Type`
string (read-only)

Type of graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For stemseries objects, `Type` is '`hggroup`'. The following statement finds all the hggroup objects in the current axes.

```
t = findobj(gca,'Type','hggroup');
```

`UIContextMenu`
handle of uicontextmenu object

Associate context menu with object. Handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object. The default value is an empty array.

`UserData`
array

User-specified data. Data you want to associate with this object (including cell arrays and structures). The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`
`{on} | off`

Visibility of object and its children.

- `on` — Object and all children of the object are visible unless the child object's `Visible` property is `off`.
- `off` — Object not displayed. However, the object still exists and you can set and query its properties.

`XData`

array

X-coordinates of scatter markers. The `scatter` function draws individual markers at each *x*-axis location in the `XData` array. The input argument `X` in the `scatter` function calling syntax assigns values to `XData`.

`XDataSource`

MATLAB variable, as a string

Link `XData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `XData`. The default value is an empty array.

```
set(h,'XDataSource','xdatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `XDataSource` does not change the object's `XData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`YData`

scalar | vector | matrix

Y-coordinates of scatter markers. The `scatter` function draws individual markers at each *y*-axis location in the `YData` array.

The input argument `Y` in the `scatter` function calling syntax assigns values to `YData`.

`YDataSource`

MATLAB variable, as a string

Link `YData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `YData`. The default value is an empty array.

```
set(h,'YDataSource','Ydatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `YDataSource` does not change the object's `YData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`ZData`

vector

Z-coordinates. A vector defining the *z*-coordinates for the graph. `XData` and `YData` must be the same length and have the same number of rows.

`ZDataSource`
MATLAB variable, as a string

Link `ZData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `ZData`. The default value is an empty array.

```
set(h,'ZDataSource','zdatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `ZDataSource` does not change the object's `ZData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Purpose Schur decomposition

Syntax

```
T = schur(A)
T = schur(A,flag)
[U,T] = schur(A,...)
```

Description The `schur` command computes the Schur form of a matrix.

`T = schur(A)` returns the Schur matrix `T`.

`T = schur(A,flag)` for real matrix A, returns a Schur matrix `T` in one of two forms depending on the value of `flag`:

`'complex'`	`T` is triangular and is complex if `A` is real and has complex eigenvalues.
`'real'`	`T` has the real eigenvalues on the diagonal and the complex eigenvalues in 2-by-2 blocks on the diagonal. `'real'` is the default when `A` is real.

If `A` is complex, `schur` returns the complex Schur form in matrix `T` and `flag` is ignored. The complex Schur form is upper triangular with the eigenvalues of `A` on the diagonal.

The function `rsf2csf` converts the real Schur form to the complex Schur form.

`[U,T] = schur(A,...)` also returns a unitary matrix `U` so that `A = U*T*U'` and `U'*U = eye(size(A))`.

Examples `H` is a 3-by-3 eigenvalue test matrix:

```
H = [ -149    -50   -154
       537    180    546
       -27     -9    -25 ]
```

Its Schur form is

```
schur(H)
```

```
ans =
     1.0000   -7.1119 -815.8706
          0    2.0000  -55.0236
          0         0    3.0000
```

The eigenvalues, which in this case are 1, 2, and 3, are on the diagonal. The fact that the off-diagonal elements are so large indicates that this matrix has poorly conditioned eigenvalues; small changes in the matrix elements produce relatively large changes in its eigenvalues.

See Also

eig | hess | qz | rsf2csf

Purpose

Sequence of MATLAB statements in file

Description

A script file is an external file that contains a sequence of MATLAB statements. By typing the filename, you can obtain subsequent MATLAB input from the file. Script files have a filename extension of `.m`.

Scripts are the simplest kind of MATLAB program. They are useful for automating blocks of MATLAB commands, such as computations you have to perform repeatedly from the command line. Scripts can operate on existing data in the workspace, or they can create new data on which to operate. Although scripts do not return output arguments, any variables that they create remain in the workspace, so you can use them in further computations. In addition, scripts can produce graphical output using commands like `plot`.

Scripts can contain any series of MATLAB statements. They require no declarations or begin/end delimiters.

Like any MATLAB program, scripts can contain comments. Any text following a percent sign (`%`) on a given line is comment text. Comments can appear on lines by themselves, or you can append them to the end of any executable line.

See Also

`echo` | `function` | `type`

Purpose

Scattered data interpolation

Description

Use `scatteredInterpolant` to perform interpolation on a 2-D or 3-D "Scattered Data" on page 1-4658 set. For example, you can pass a set of `(x,y)` points and values, `v`, to `scatteredInterpolant`, and it returns a surface of the form $v = F(x, y)$. This surface always passes through the sample values at the point locations. You can evaluate this surface at any query point, `(xq,yq)`, to produce an interpolated value, `vq`.

Use `scatteredInterpolant` to create the "Interpolant" on page 1-4658, `F`. Then, you can evaluate `F` at specific points using any of the following syntaxes:

- `Vq = F(Pq)` specifies the query points in the matrix `Pq`. Each row in `Pq` contains the coordinates of a query point.
- `Vq = F(Xq,Yq)` and `Vq = F(Xq,Yq,Zq)` specify the query points as two or three matrices of equal size.
- `Vq = F({xq,yq})` and `Vq = F({xq,yq,zq})` specify the query points as "Grid Vectors" on page 1-4658. The interpolated values are returned in `Vq`. Use this syntax to conserve memory when you want to query a large grid of points.

Construction

`F = scatteredInterpolant(x,y,v)` creates an interpolant that fits a surface of the form $v = F(x,y)$. Vectors `x` and `y` specify the `(x,y)` coordinates of the sample points. `v` is a vector that contains the sample values associated with the points, `(x,y)`.

`F = scatteredInterpolant(x,y,z,v)` creates a 3-D interpolant of the form $v = F(x,y,z)$.

`F = scatteredInterpolant(P,v)` specifies the coordinates of the sample points as an array. The rows of `P` contain the (x, y) or (x, y, z) coordinates for the values in `v`.

`F = scatteredInterpolant(___,Method)` specifies a string that describes an interpolation method: `'nearest'`, `'linear'`, or `'natural'`. Specify `Method` as the last input argument in any of the first three syntaxes.

`F = scatteredInterpolant(___,Method,ExtrapolationMethod)` specifies both the interpolation and extrapolation methods as strings. `Method` can be one of three strings: `'nearest'`, `'linear'`, or `'natural'`. Specify `ExtrapolationMethod` as one of the following strings: `'nearest'`, `'linear'`, or `'none'`. Pass `Method` and `ExtrapolationMethod` together as the last two input arguments in any of the first three syntaxes.

`F = scatteredInterpolant()` creates an empty scattered data interpolant. Use `F.Points = P` to initialize `F` with the points in matrix `P`. Use `F.Values = v` to initialize `F` with the values in `v`.

Input Arguments

x

Sample points *x*-coordinates, specified as a vector of the same size as `v`.

y

Sample points *y*-coordinates, specified as a vector of the same size as `v`.

z

Sample points *z*-coordinates, specified as a vector of the same size as `v`.

P

Sample points array, specified as an `m`-by-`n` matrix, where `m` is the number of points and `n` is the dimension of the space where the points reside. Each row of `P` contains the (x, y) or (x, y, z) coordinates of a sample point.

v

Sample values vector, specified as a vector of that defines the values at the sample points.

Method

Interpolation method, specified as one of these strings.

Method String	Description
`'linear'` (default)	Linear interpolation.
`'nearest'`	Nearest neighbor interpolation.
`'natural'`	Natural neighbor interpolation.

ExtrapolationMethod

Extrapolation method, specified as one of these strings.

ExtrapolationMethod String	Description
`'linear'`	Linear extrapolation based on boundary gradients. Default when `Method` is
`'nearest'`	Nearest neighbor extrapolation. This method evaluates to the value of the nearest neighbor on the boundary. Default when `Method = 'nearest'`.
`'none'`	No extrapolation. Any queries outside the convex hull of `F.Points` return `NaN`.

Properties

Points

Array of sample points (locations) for the values in `F.Values`. Each row of `F.Points` contains the (*x*, *y*) or (*x*, *y*, *z*) coordinates of a sample point.

Values

Vector of values associated with each point in `F.Points`.

Method

A string specifying the method used to interpolate the data: `'nearest'`, `'linear'`, or `'natural'`.

ExtrapolationMethod

A string specifying the method used to extrapolate the data: `'nearest'`, `'linear'`, or `'none'`. A value of `'none'` indicates that extrapolation is disabled.

Definitions

Interpolant

Interpolating function that you can evaluate at query locations.

Scattered Data

A set of points that have no structure among their relative locations.

Full Grid

A grid represented as a set of arrays. For example, you can create a full grid using `ndgrid`.

Grid Vectors

A set of vectors that serve as a compact representation of a grid in `ndgrid` format. For example, `[X,Y] = ndgrid(xg,yg)` returns a full grid in the matrices `X` and `Y`. You can represent the same grid using the grid vectors, `xg` and `yg`.

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

Indexing

`scatteredInterpolant` supports index-based editing of the properties of `F`. You can add or remove points in `F.Points` and update the corresponding values in `F.Values`. For example, `F.Points(5,:) = []` removes the fifth point, and `F.Values(5) = []` removes the corresponding value.

Examples

2-D Interpolation

Define 200 random points.

```
xy = -2.5 + 5*gallery('uniformdata',[200 2],0);
x = xy(:,1);
```

```
y = xy(:,2);
```

Sample an exponential function. These are the sample values for the interpolant.

```
v = x.*exp(-x.^2-y.^2);
```

Create the interpolant.

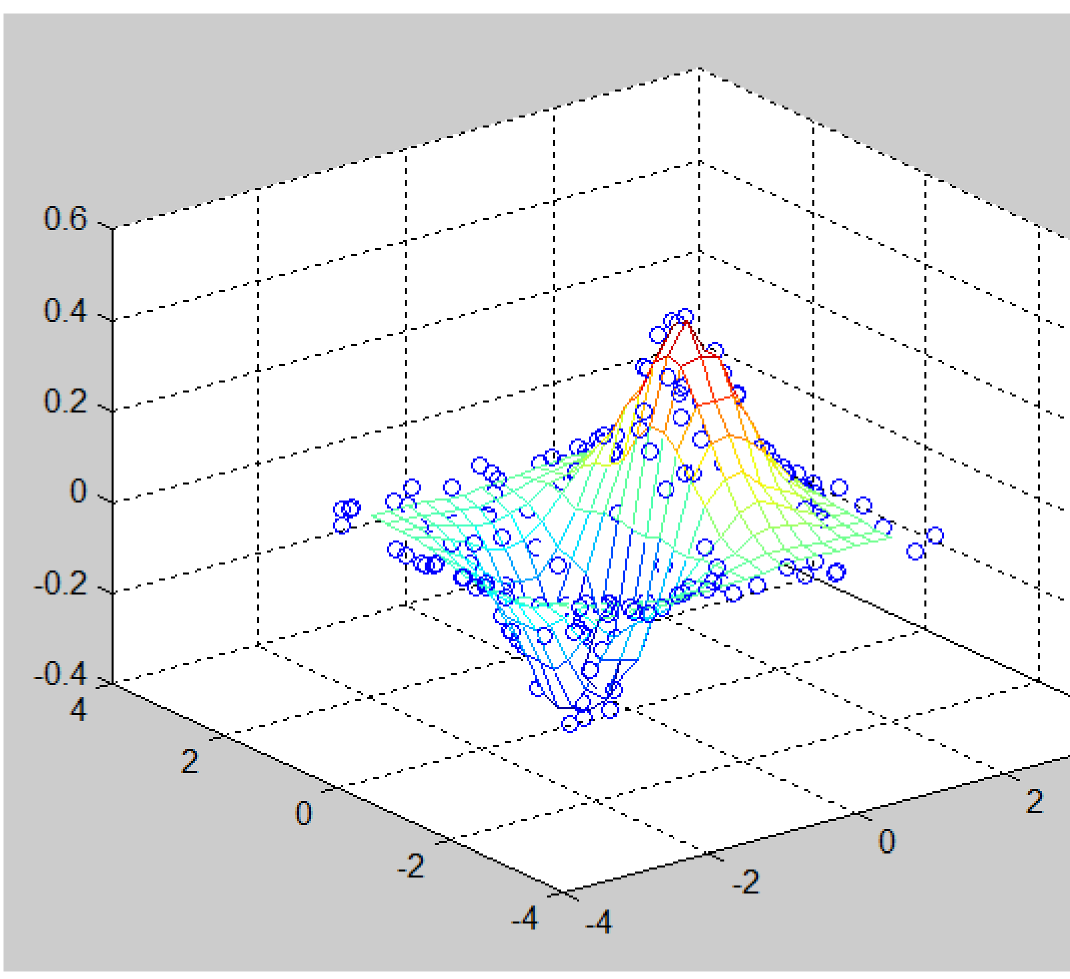
```
F = scatteredInterpolant(x,y,v);
```

Evaluate the interpolant at query locations (*xq*, *yq*).

```
ti = -2:.25:2;
[xq,yq] = meshgrid(ti,ti);
vq = F(xq,yq);
```

Plot the result.

```
figure
mesh(xq,yq,vq);
hold on;
plot3(x,y,v,'o');
hold off;
```



2-D Extrapolation

Query the interpolant at a single point outside the convex hull using nearest neighbor extrapolation.

Define a matrix of 200 random points.

```
P = -2.5 + 5*gallery('uniformdata',[200 2],0);
```

Sample an exponential function. These are the sample values for the interpolant.

```
x = P(:,1);
y = P(:,2);
v = x.*exp(-x.^2-y.^2);
```

Create the interpolant, specifying linear interpolation and nearest neighbor extrapolation.

```
F = scatteredInterpolant(P,v,'linear','nearest')
```

```
F =

  scatteredInterpolant with properties:

                 Points: [200x2 double]
                 Values: [200x1 double]
                 Method: 'linear'
    ExtrapolationMethod: 'nearest'
```

Evaluate the interpolant outside the convex hull.

```
vq = F(3.0,-1.5)
```

```
vq =

    0.0031
```

Disable extrapolation and evaluate F at the same point.

```
F.ExtrapolationMethod = 'none';
vq = F(3.0,-1.5)
```

```
vq =

   NaN
```

Replacement of Sample Values

Replace the elements in the `Values` property when you want to change the values at the sample points. You get immediate results when you evaluate the new interpolant because the original triangulation has not changed.

Create 50 random points.

```
x = -2.5 + 5*gallery('uniformdata',[50 1],0);
y = -2.5 + 5*gallery('uniformdata',[50 1],1);
```

Sample an exponential function. These are the sample values for the interpolant.

```
v = x.*exp(-x.^2-y.^2);
```

Create the interpolant.

```
F = scatteredInterpolant(x,y,v)
```

```
F =

  scatteredInterpolant with properties:

                 Points: [50x2 double]
                 Values: [50x1 double]
                 Method: 'linear'
    ExtrapolationMethod: 'linear'
```

Evaluate the interpolant at `(1.40,1.90)`.

```
F(1.40,1.90)
```

```
ans =

    0.0029
```

Change the interpolant sample values.

```
vnew = x.^2 + y.^2;
F.Values = vnew;
```

Evaluate the interpolant at (1.40,1.90).

```
F(1.40,1.90)
```

```
ans =

    6.1109
```

See Also

griddedInterpolant | interp1 | interp2 | interp3 | meshgrid | ndgrid

How To

- Class Attributes
- Property Attributes
- "Interpolating Gridded Data"

Purpose

Secant of argument in radians

Syntax

```
Y = sec(X)
```

Description

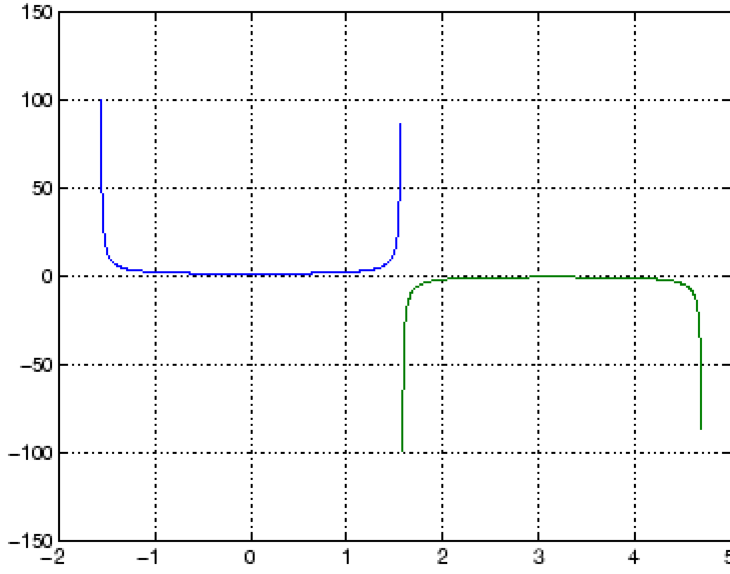
The `sec` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = sec(X)` returns an array the same size as `X` containing the secant of the elements of `X`.

Examples

Graph the secant over the domains $-\pi/2 < x < \pi/2$ and $\pi/2 < x < 3\pi/2$.

```
x1 = -pi/2+0.01:0.01:pi/2-0.01;
x2 = pi/2+0.01:0.01:(3*pi/2)-0.01;
plot(x1,sec(x1),x2,sec(x2)), grid on
```



The expression `sec(pi/2)` does not evaluate as infinite but as the reciprocal of the floating-point accuracy `eps`, because `pi` is a floating-point approximation to the exact value of π.

See Also

secd | sech | asec | asecd | asech

secd

Purpose

Secant of argument in degrees

Syntax

```
Y = secd(X)
```

Description

`Y = secd(X)` returns the secant of the elements of `X`, which are expressed in degrees.

Input Arguments

X - Angle in degrees

scalar value | vector | matrix | N-D array

Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `secd` operation is element-wise when `X` is nonscalar.

Example:

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

Output Arguments

Y - Secant of angle

scalar value | vector | matrix | N-D array

Secant of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as `X`.

Examples

Secant of 90 degrees compared to secant of /2 radians

`secd(90)` is infinite, whereas `sec(pi/2)` is large but finite.

```
secd(90)

ans =

   Inf

sec(pi/2)

ans =
```

```
   1.6331e+16
```

Secant of vector of complex angles, specified in degrees

```
z = [35+i 15+2i 10+3i];
y = secd(z)

y =

   1.2204 + 0.0149i   1.0346 + 0.0097i   1.0140 + 0.0094i
```

See Also

sec | asec | asecd

sech

Purpose

Hyperbolic secant

Syntax

```
Y = sech(X)
```

Description

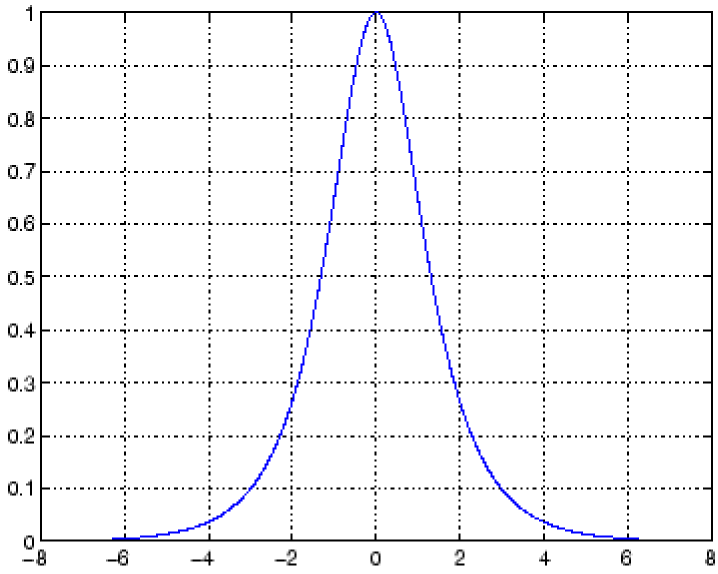
The `sech` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = sech(X)` returns an array the same size as `X` containing the hyperbolic secant of the elements of `X`.

Examples

Graph the hyperbolic secant over the domain $-2\pi \leq x \leq 2\pi$.

```
x = -2*pi:0.01:2*pi;
plot(x,sech(x)), grid on
```



See Also

`asech` | `sec` | `sinh` | `cosh`

Purpose

Select, move, resize, or copy axes and uicontrol graphics objects

Syntax

```
A = selectmoveresize
set(gca,'ButtonDownFcn','selectmoveresize')
```

Description

`selectmoveresize` is useful as the callback routine for axes and uicontrol button down functions. When executed, it selects the object and allows you to move, resize, and copy it.

`A = selectmoveresize` returns a structure array containing

- `A.Type`: a string containing the action type, which can be `Select`, `Move`, `Resize`, or `Copy`
- `A.Handles`: a list of the selected handles, or, for a `Copy`, an m-by-2 matrix containing the original handles in the first column and the new handles in the second column

`set(gca,'ButtonDownFcn','selectmoveresize')` sets the `ButtonDownFcn` property of the current axes to selectmoveresize:

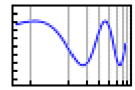
See Also

`axes` | `uicontrol`

semilogx

Purpose

Semilogarithmic plot



Syntax

```
semilogx(Y)
semilogx(X1,Y1,...)
semilogx(X1,Y1,LineSpec,...)
semilogx(...,'PropertyName',PropertyValue,...)
h = semilogx(...)
```

Description

`semilogx` plot data as logarithmic scales for the *x*-axis.

`semilogx(Y)` creates a plot using a base 10 logarithmic scale for the *x*-axis and a linear scale for the *y*-axis. It plots the columns of `Y` versus their index if `Y` contains real numbers. `semilogx(Y)` is equivalent to `semilogx(real(Y), imag(Y))` if `Y` contains complex numbers. `semilogx` ignores the imaginary component in all other uses of this function.

`semilogx(X1,Y1,...)` plots all `Xn` versus `Yn` pairs. If only `Xn` or `Yn` is a matrix, `semilogx` plots the vector argument versus the rows or columns of the matrix, along the dimension of the matrix whose length matches the length of the vector. If the matrix is square, its columns plot against the vector if their lengths match.

`semilogx(X1,Y1,LineSpec,...)` plots all lines defined by the `Xn,Yn,LineSpec` triples. `LineSpec` determines line style, marker symbol, and color of the plotted lines.

`semilogx(...,'PropertyName',PropertyValue,...)` sets property values for all `lineseries properties` graphics objects created by `semilogx`.

`h = semilogx(...)` return a vector of handles to `lineseries` graphics objects, one handle per line.

Tips

If you do not specify a color when plotting more than one line, `semilogx` automatically cycle through the colors and line styles in the order specified by the current axes `ColorOrder` and `LineStyleOrder` properties.

You can mix `Xn,Yn` pairs with `Xn,Yn,`*`LineSpec`* triples; for example,

```
semilogx(X1,Y1,X2,Y2,LineSpec,X3,Y3)
```

If you attempt to add a `loglog`, `semilogx`, or `semilogy` plot to a linear axis mode graph with `hold on`, the axis mode remains as it is and the new data plots as linear.

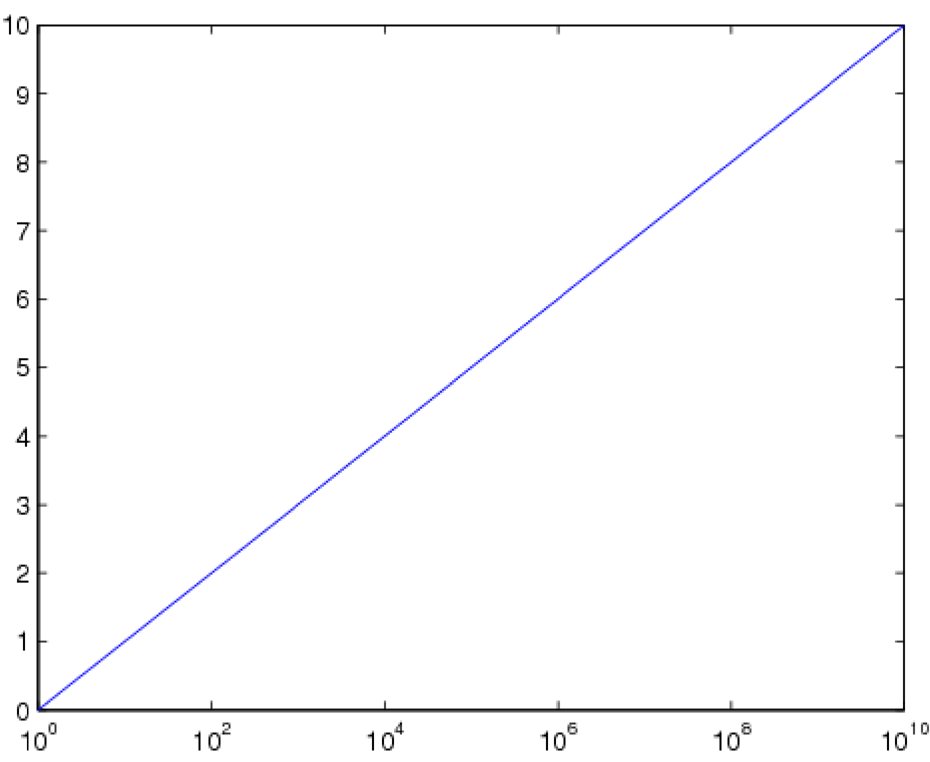
Renderer Support

The OpenGL renderer does not support logarithmic-scale axes. MATLAB automatically selects a different renderer when using logarithmic scaling. If you set the figure `Renderer` property to `opengl`, axis scales become linear. See the figure Renderer property for more information on renderers.

Examples

Create a simple `semilogx` plot.

```
x = 0:0.1:10;
semilogx(10.^x,x)
```

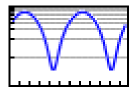


See Also

`line` | `LineSpec` | `loglog` | `plot` | `semilogy`

Purpose

Semilogarithmic plot



Syntax

```
semilogy(Y)
semilogy(X1,Y1,...)
semilogy(X1,Y1,LineSpec,...)
semilogy(...,'PropertyName',PropertyValue,...)
h = semilogy(...)
```

Description

`semilogy` plots data with logarithmic scale for the *y*-axis.

`semilogy(Y)` creates a plot using a base 10 logarithmic scale for the *y*-axis and a linear scale for the *x*-axis. It plots the columns of `Y` versus their index if `Y` contains real numbers. `semilogy(Y)` is equivalent to `semilogy(real(Y), imag(Y))` if `Y` contains complex numbers. `semilogy` ignores the imaginary component in all other uses of this function.

`semilogy(X1,Y1,...)` plots all `Xn` versus `Yn` pairs. If only `Xn` or `Yn` is a matrix, `semilogy` plots the vector argument versus the rows or columns of the matrix, along the dimension of the matrix whose length matches the length of the vector. If the matrix is square, its columns plot against the vector if their lengths match.

`semilogy(X1,Y1,LineSpec,...)` plots all lines defined by the `Xn,Yn,LineSpec` triples. `LineSpec` determines line style, marker symbol, and color of the plotted lines.

`semilogy(...,'PropertyName',PropertyValue,...)` sets property values for all `lineseries properties` graphics objects created by `semilogy`.

`h = semilogy(...)` returns a vector of handles to `lineseries` graphics objects, one handle per line.

Tips

If you do not specify a color when plotting more than one line, `semilogy` automatically cycle through the colors and line styles in the order specified by the current axes `ColorOrder` and `LineStyleOrder` properties.

You can mix `Xn,Yn` pairs with `Xn,Yn,`*`LineSpec`* triples; for example,

```
semilogy(X1,Y1,X2,Y2,LineSpec,X3,Y3)
```

If you attempt to add a `loglog`, `semilogx`, or `semilogy` plot to a linear axis mode graph with `hold on`, the axis mode remains as it is and the new data plots as linear.

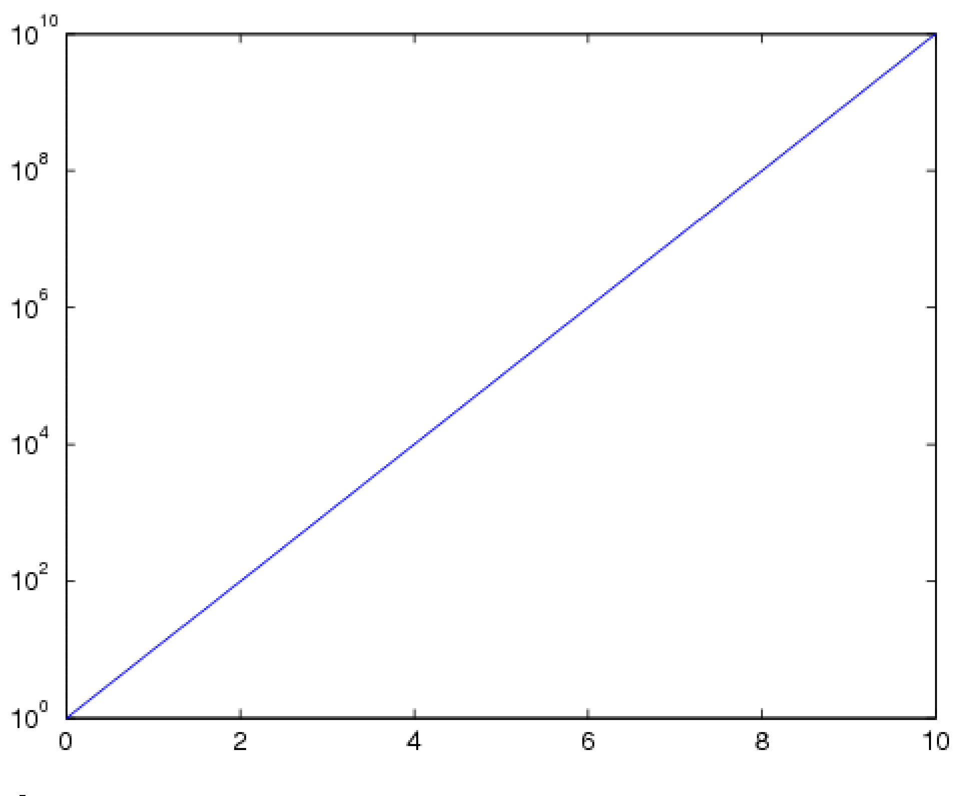
Renderer Support

The OpenGL renderer does not support logarithmic-scale axes. MATLAB automatically selects a different renderer when using logarithmic scaling. If you set the figure `Renderer` property to `opengl`, axis scales become linear. See the figure Renderer property for more information on renderers.

Examples

Create a simple `semilogy` plot.

```
x = 0:.1:10;
semilogy(x,10.^x)
```



See Also

line | LineSpec | loglog | plot | semilogx

sendmail

Purpose

Send email message to address list

Syntax

```
sendmail(recipients,subject)
sendmail(recipients,subject,message)
sendmail(recipients,subject,message,attachments)
```

Description

`sendmail(recipients,subject)` sends email to `recipients` with the specified `subject`. The `recipients` input is a string for a single address, or a cell array of strings for multiple addresses.

`sendmail(recipients,subject,message)` includes the specified `message`. If `message` is a string, `sendmail` automatically wraps text at 75 characters. To force a line break in the message text, use `10`, as shown in the Examples. If `message` is a cell array of strings, each cell represents a new line of text.

`sendmail(recipients,subject,message,attachments)` attaches the files listed in the string or cell array `attachments`.

Tips

- The `sendmail` function does not support HTML-formatted messages. However, you can send HTML files as attachments.
- If `sendmail` cannot determine your email address or outgoing SMTP mail server from your system registry, specify those settings using the `setpref` function. For example:

  ```
  setpref('Internet','SMTP_Server','my_server.example.com');
  setpref('Internet','E_mail','my_email@example.com');
  ```

  To identify the SMTP server for the call to `setpref`, check the preferences for your electronic mail application, or consult your email system administrator. If you cannot easily determine the server name, try `'mail'`, which is a common default, such as:

  ```
  setpref('Internet','SMTP_Server','mail');
  ```

- By default, the `sendmail` function does not support email servers that require authentication. To support these servers, change your

system settings and set preferences for the SMTP user name and password, with commands in the following form:

```
props = java.lang.System.getProperties;
props.setProperty('mail.smtp.auth','true');

setpref('Internet','SMTP_Username','myaddress@example.com');
setpref('Internet','SMTP_Password','mypassword');
```

Examples

Send a message with two attachments to a hypothetical email address:

```
sendmail('user@otherdomain.com',...
         'Test subject','Test message',...
         {'folder/attach1.html','attach2.doc'});
```

Send a message with forced line breaks (using 10) to a hypothetical email address:

```
sendmail('user@otherdomain.com','New subject', ...
        ['Line1 of message' 10 'Line2 of message' 10 ...
         'Line3 of message' 10 'Line4 of message']);
```

The resulting message is:

```
Line1 of message
Line2 of message
Line3 of message
Line4 of message
```

Gmail™ servers require authentication and an encrypted connection (SSL) on port 465. Change your system settings to use your Gmail server and send a test message to your Gmail account:

```
% Modify these two lines to reflect
% your account and password.
```

```
myaddress = 'myaddress@gmail.com';
mypassword = 'mypassword';

setpref('Internet','E_mail',myaddress);
setpref('Internet','SMTP_Server','smtp.gmail.com');
setpref('Internet','SMTP_Username',myaddress);
setpref('Internet','SMTP_Password',mypassword);

props = java.lang.System.getProperties;
props.setProperty('mail.smtp.auth','true');
props.setProperty('mail.smtp.socketFactory.class', ...
                  'javax.net.ssl.SSLSocketFactory');
props.setProperty('mail.smtp.socketFactory.port','465');

sendmail(myaddress, 'Gmail Test', 'This is a test message.');
```

Alternatives

On Windows systems with Microsoft Outlook®, you can send email directly through Outlook by accessing the COM server with `actxserver`. For an example, see Solution 1-RTY6J.

See Also

`getpref` | `setpref`

How To

- “Specify Proxy Server Settings for Connecting to the Internet”

Purpose

Create serial port object

Syntax

```
obj = serial('port')
obj = serial('port','PropertyName',PropertyValue,...)
```

Description

`obj = serial('port')` creates a serial port object associated with the serial port specified by *`port`*. If *`port`* does not exist, or if it is in use, you will not be able to connect the serial port object to the device.

`Port` object name will depend upon the platform that the serial port is on. `instrhwinfo` ('serial') provides a list of available serial ports. This list is an example of serial constructors on different platforms:

Platform	Serial Port Constructor
Linux and Linux 64	`serial('/dev/ttyS0');`
Mac OS X 64	`serial('/dev/tty.KeySerial1');`
Windows 32 and Windows 64	`serial('com1');`

`obj = serial('port','PropertyName',PropertyValue,...)` creates a serial port object with the specified property names and property values. If an invalid property name or property value is specified, an error is returned and the serial port object is not created.

Tips

When you create a serial port object, these property values are automatically configured:

- The `Type` property is given by `serial`.
- The `Name` property is given by concatenating `Serial` with the port specified in the `serial` function.
- The `Port` property is given by the port specified in the `serial` function.

You can specify the property names and property values using any format supported by the `set` function. For example, you can use property name/property value cell array pairs. Additionally, you can

specify property names without regard to case, and you can make use of property name completion. For example, the following commands are all valid on a Windows platform.

```
s = serial('COM1','BaudRate',4800);
s = serial('COM1','baudrate',4800);
s = serial('COM1','BAUD',4800);
```

Refer to Configuring Property Values for a list of serial port object properties that you can use with `serial`.

Before you can communicate with the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt a read or write operation while the object is not connected to the device. You can connect only one serial port object to a given serial port.

Examples

This example creates the serial port object `s1` associated with the serial port COM1 on a Windows platform.

```
s1 = serial('COM1');
```

The `Type`, `Name`, and `Port` properties are automatically configured.

```
get(s1,{'Type','Name','Port'})
ans =
    'serial'    'Serial-COM1'    'COM1'
```

To specify properties during object creation

```
s2 = serial('COM2','BaudRate',1200,'DataBits',7);
```

See Also

`fclose` | `fopen` | `Name` | `Port` | `Status` | `Type`

Purpose

Send break to device connected to serial port

Syntax

```
serialbreak(obj)
serialbreak(obj,time)
```

Description

`serialbreak(obj)` sends a break of 10 milliseconds to the device connected to the serial port object, `obj`.

`serialbreak(obj,time)` sends a break to the device with a duration, in milliseconds, specified by `time`. Note that the duration of the break might be inaccurate under some operating systems.

Tips

For some devices, the break signal provides a way to clear the hardware buffer.

Before you can send a break to the device, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to send a break while `obj` is not connected to the device.

`serialbreak` is a synchronous function, and blocks the command line until execution is complete.

If you issue `serialbreak` while data is being asynchronously written, an error is returned. In this case, you must call the `stopasync` function or wait for the write operation to complete.

See Also

`fopen` | `stopasync` | `Status`

Purpose

Set Handle Graphics object properties

Syntax

```
set(H,'PropertyName',PropertyValue,...)
set(H,a)
set(H,pn,pv,...)
set(H,pn,MxN_pv)
a = set(h)
pv = set(h,'PropertyName')
```

Description

Note Do not use the `set` function on Java objects as it will cause a memory leak. For more information, see "Accessing Private and Public Data"

`set(H,'PropertyName',PropertyValue,...)` sets the named properties to the specified values on the object(s) identified by `H`. `H` can be a vector of handles, in which case `set` sets the properties' values for all the objects.

`set(H,a)` sets the named properties to the specified values on the object(s) identified by `H`. `a` is a structure array whose field names are the object property names and whose field values are the values of the corresponding properties.

`set(H,pn,pv,...)` sets the named properties specified in the cell array `pn` to the corresponding value in the cell array `pv` for all objects identified in `H`.

`set(H,pn,MxN_pv)` sets `n` property values on each of `m` graphics objects, where `m = length(H)` and `n` is equal to the number of property names contained in the cell array `pn`. This allows you to set a given group of properties to different values on each object.

`a = set(h)` returns the user-settable properties and possible values for the object identified by `h`. `a` is a structure array whose field names are the object's property names and whose field values are the possible values of the corresponding properties. If you do not specify an output

argument, the MATLAB software displays the information on the screen. `h` must be scalar.

`pv = set(h,'PropertyName')` returns the possible values for the named property. If the possible values are strings, `set` returns each in a cell of the cell array `pv`. For other properties, `set` returns a statement indicating that `PropertyName` does not have a fixed set of property values. If you do not specify an output argument, MATLAB displays the information on the screen. `h` must be scalar.

Tips

You can use any combination of property name/property value pairs, structure arrays, and cell arrays in one call to `set`.

Setting Property Units

Note that if you are setting both the `FontSize` and the `FontUnits` properties in one function call, you must set the `FontUnits` property first so that the MATLAB software can correctly interpret the specified `FontSize`. The same applies to figure and axes uints — always set the `Units` property before setting properties whose values you want to be interpreted in those units. For example,

```
f = figure('Units','characters',...
        'Position',[30 30 120 35]);
```

Examples

Set the Color property of the current axes to blue.

```
axes;
set(gca,'Color','b')
```

Change all the `lines` in a plot to black.

```
plot(peaks)
set(findobj('Type','line'),'Color','k')
```

You can define a group of properties in a structure to better organize your code. For example, these statements define a structure called `active`, which contains a set of property definitions used for the

`uicontrol` objects in a particular figure. When this figure becomes the current figure, MATLAB changes the colors and enables the controls.

```
active.BackgroundColor = [.7 .7 .7];
active.Enable = 'on';
active.ForegroundColor = [0 0 0];

if gcf == control_fig_handle
 set(findobj(control_fig_handle,'Type','uicontrol'),active)
end
```

You can use cell arrays to set properties to different values on each object. For example, these statements define a cell array to set three properties,

```
PropName(1) = {'BackgroundColor'};
PropName(2) = {'Enable'};
PropName(3) = {'ForegroundColor'};
```

These statements define a cell array containing three values for each of three objects (i.e., a 3-by-3 cell array).

```
PropVal(1,1) = {[.5 .5 .5]};
PropVal(1,2) = {'off'};
PropVal(1,3) = {[.9 .9 .9]};
PropVal(2,1) = {[1 0 0]};
PropVal(2,2) = {'on'};
PropVal(2,3) = {[1 1 1]};
PropVal(3,1) = {[.7 .7 .7]};
PropVal(3,2) = {'on'};
PropVal(3,3) = {[0 0 0]};
```

Now pass the arguments to `set`,

```
set(H,PropName,PropVal)
```

where `length(H) = 3` and each element is the handle to a uicontrol.

Setting Different Values for the Same Property on Multiple Objects

Suppose you want to set the value of the `Tag` property on five line objects, each to a different value. Note how the value cell array needs to be transposed to have the proper shape.

```
h = plot(rand(5));
set(h,{'Tag'},{'line1','line2','line3','line4','line5'}')
```

See Also

`findobj` | `gca` | `gcf` | `gco` | `gcbo` | `get`

audioplayer.set

Purpose

Set property values for `audioplayer` object

Syntax

```
set(obj,Name,Value)
set(obj,cellOfNames,cellOfValues)
set(obj,structOfProperties)
settableProperties = set(obj)
```

Description

`set(obj,Name,Value)` sets the named property to the specified value for the object `obj`.

`set(obj,cellOfNames,cellOfValues)` sets the properties listed in the cell array `cellOfNames` to the corresponding values in the cell array `cellOfValues`. Each cell array must contain the same number of elements.

`set(obj,structOfProperties)` sets the properties identified by each field of the structure array `structOfProperties` to the values of the associated fields.

`settableProperties = set(obj)` returns the names of the properties that you can set in a structure array. The field names of `settableProperties` are the property names.

Tips

The `set` function allows combinations of property name/value pairs, cell array pairs, and structure arrays in the same function call.

Examples

View the list of properties that you can set for an `audioplayer` object:

```
load handel.mat;
handelObj = audioplayer(y, Fs);
set(handelObj)
```

Set the `Tag` and `UserData` properties of an `audioplayer` object using a structure array:

```
newValues.Tag = 'My Tag';
newValues.UserData = {'My User Data', pi, [1 2 3 4]};
```

```
load handel.mat;
handelObj = audioplayer(y, Fs);
set(handelObj, newValues)

% View the values all properties.
get(handelObj)
```

Alternatives

To set the value of a single property, you can use dot notation. Reference each property as though it is a field of a structure array. For example, set the `Tag` property for an object called `handelObj` (as created in the Examples):

```
handelObj.Tag = 'This is my tag.';
```

This command is exactly equivalent to:

```
set(handelObj, 'Tag', 'This is my tag.');
```

See Also

`audioplayer` | `get`

audiorecorder.set

Purpose

Set property values for `audiorecorder` object

Syntax

```
set(obj,Name,Value)
set(obj,cellOfNames,cellOfValues)
set(obj,structOfProperties)
settableProperties = set(obj)
```

Description

`set(obj,Name,Value)` sets the named property to the specified value for the object *`obj`*.

`set(obj,cellOfNames,cellOfValues)` sets the properties listed in the cell array *`cellOfNames`* to the corresponding values in the cell array *`cellOfValues`*. Each cell array must contain the same number of elements.

`set(obj,structOfProperties)` sets the properties identified by each field of the structure array *`structOfProperties`* to the values of the associated fields.

`settableProperties = set(obj)` returns the names of the properties that you can set in a structure array. The field names of *`settableProperties`* are the property names.

Tips

The `set` function allows combinations of property name/value pairs, cell array pairs, and structure arrays in the same function call.

Examples

View the list of properties that you can set for an `audiorecorder` object:

```
recorderObj = audiorecorder;
set(recorderObj)
```

Set the `Tag` and `UserData` properties of an `audiorecorder` object using a structure array:

```
newValues.Tag = 'My Tag';
newValues.UserData = {'My User Data', pi, [1 2 3 4]};
```

```
recorderObj = audiorecorder;
set(recorderObj, newValues)

% View the values all properties.
get(recorderObj)
```

Alternatives

To set the value of a single property, you can use dot notation. Reference each property as though it is a field of a structure array. For example, set the `Tag` property for an object called `recorderObj` (as created in the Examples):

```
recorderObj.Tag = 'This is my tag.';
```

This command is exactly equivalent to:

```
set(recorderObj, 'Tag', 'This is my tag.');
```

See Also

`audiorecorder` | `get`

set (COM)

Purpose

Set object or interface property to specified value

Syntax

```
h.set('pname',value)
h.set('pname1',value1,'pname2',value2,...)
set(h,...)
```

Description

`h.set('pname',value)` sets the property specified in the string `pname` to the given `value`.

`h.set('pname1',value1,'pname2',value2,...)` sets each property specified in the `pname` strings to the given `value`.

`set(h,...)` is an alternate syntax for the same operation.

For information on how MATLAB converts workspace matrices to COM data types, see "Handling COM Data in MATLAB Software".

Tips

COM functions are available on Microsoft Windows systems only.

Examples

Create an `mwsamp` control and use `set` to change the `Label` and `Radius` properties:

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.1', [0 0 200 200], f);

h.set('Label', 'Click to fire event', 'Radius', 40);
h.invoke('Redraw');
```

Here is another way to do the same thing, only without `set` and `invoke`:

```
h.Label = 'Click to fire event';
h.Radius = 40;
h.Redraw;
```

See Also

`get (COM)` | `inspect` | `isprop` | `addproperty` | `deleteproperty`

Purpose

Assign property values to handle objects derived from `hgsetget` class

Syntax

```
set(H,'PropertyName',value,...)
set(H,pn,pv)
set(H,S)
S = set(h)
```

Description

`set(H,'PropertyName',value,...)` sets the named property to the specified value for the objects in the handle array `H`.

`set(H,pn,pv)` sets the named properties specified in the cell array of strings `pn` to the corresponding values in the cell array `pv` for all objects specified in `H`. The cell array `pn` must be `1`-by-`n` (where `n` is the number of property names), but the cell array `pv` can be `m`-by-`n` where `m` is equal to `length(H)`. `set` updates each object with the associated set of values for the list of property names contained in.

`set(H,S)` sets the properties identified by each field name of `struct` `S` with the values contained in `S`. `S` is a `struct` whose field names are object property names.

`S = set(h)` returns the user-settable properties of scalar `h`. `S` is a `struct` whose field names are the object's property names and whose values are empty cell arrays.

Override the `hgsetget` class `setdisp` method to change how MATLAB displays this information.

See Also

`handle` | `hgsetget` | `set` | `get (hgsetget)`

How To

- "Implementing a Set/Get Interface for Properties"

VideoReader.set

Purpose

Set property values for video reader object

Syntax

```
set(obj,Name,Value)
set(obj,cellOfNames,cellOfValues)
set(obj,structOfProperties)
settableProperties = set(obj)
```

Description

`set(obj,Name,Value)` sets the named property to the specified value for the object *obj*.

`set(obj,cellOfNames,cellOfValues)` sets the properties listed in the cell array *cellOfNames* to the corresponding values in the cell array *cellOfValues*. Each cell array must contain the same number of elements.

`set(obj,structOfProperties)` sets the properties identified by each field of the structure array *structOfProperties* to the values of the associated fields.

`settableProperties = set(obj)` returns the names of the properties that you can set in a structure array. The field names of *settableProperties* are the property names.

Tips

The `set` function allows combinations of property name/value pairs, cell array pairs, and structure arrays in the same function call.

Examples

View the list of properties that you can set for a video reader object:

```
xyloObj = VideoReader('xylophone.mpg');
set(xyloObj)
```

Set the `Tag` and `UserData` properties of a video reader object using a structure array:

```
newValues.Tag = 'My Tag';
newValues.UserData = {'My User Data', pi, [1 2 3 4]};
```

```
xyloObj = VideoReader('xylophone.mpg');
set(xyloObj, newValues)

% View the values all properties.
get(xyloObj)
```

Alternatives

To set the value of a single property, you can use dot notation. Reference each property as though it is a field of a structure array. For example, set the `Tag` property for a reader object called `xyloObj` (as created in the Examples):

```
xyloObj.Tag = 'This is my tag.';
```

This command is exactly equivalent to:

```
set(xyloObj, 'Tag', 'This is my tag.');
```

See Also

`VideoReader` | `get`

set (RandStream)

Purpose

Set random number stream property

Class

`RandStream`

Syntax

```
set(stream,'PropertyName',Value)
set(stream,'Property1',Value1,'Property2',Value2,...)
set(stream,A)
A = set(stream,'Property')
set(stream,'Property')
A = set(stream)
set(stream)
```

Description

`set(stream,'PropertyName',Value)` sets the property `'PropertyName'` of the random stream `stream` to the value `Value`.

`set(stream,'Property1',Value1,'Property2',Value2,...)` sets multiple random stream property values with a single statement.

`set(stream,A)` where `A` is a structure whose field names are property names of the random stream `stream` sets the properties of `stream` named by each field with the values contained in those fields.

`A = set(stream,'Property')` or `set(stream,'Property')` displays possible values for the specified property of `stream`.

`A = set(stream)` or `set(stream)` displays or returns all writable properties of `stream` and their possible values.

See Also

`RandStream` | `get (RandStream)` | `rand` | `randn` | `randi`

Purpose

Configure or display serial port object properties

Syntax

```
set(obj)
props = set(obj)
set(obj,'PropertyName')
props = set(obj,'PropertyName')
set(obj,'PropertyName',PropertyValue,...)
set(obj,PN,PV)
set(obj,S)
```

Description

`set(obj)` displays all configurable properties values for the serial port object, `obj`. If a property has a finite list of possible string values, then these values are also displayed.

`props = set(obj)` returns all configurable properties and their possible values for `obj` to `props`. `props` is a structure whose field names are the property names of `obj`, and whose values are cell arrays of possible property values. If the property does not have a finite set of possible values, then the cell array is empty.

`set(obj,'PropertyName')` displays the valid values for *PropertyName* if it possesses a finite list of string values.

`props = set(obj,'PropertyName')` returns the valid values for *PropertyName* `to props`. `props` is a cell array of possible string values or an empty cell array if *PropertyName* does not have a finite list of possible values.

`set(obj,'PropertyName',PropertyValue,...)` configures multiple property values with a single command.

`set(obj,PN,PV)` configures the properties specified in the cell array of strings `PN` to the corresponding values in the cell array `PV`. `PN` must be a vector. `PV` can be m-by-n where m is equal to the number of serial port objects in `obj` and n is equal to the length of `PN`.

`set(obj,S)` configures the named properties to the specified values for `obj`. `S` is a structure whose field names are serial port object properties, and whose field values are the values of the corresponding properties.

Tips

Refer to Configuring Property Values for a list of serial port object properties that you can configure with `set`.

You can use any combination of property name/property value pairs, structures, and cell arrays in one call to `set`. Additionally, you can specify a property name without regard to case, and you can make use of property name completion. For example, if `s` is a serial port object, then the following commands are all valid.

```
set(s,'BaudRate')
set(s,'baudrate')
set(s,'BAUD')
```

If you use the `help` command to display help for `set`, then you need to supply the pathname shown below.

```
help serial/set
```

Examples

This example illustrates some of the ways you can use `set` to configure or return property values for the serial port object `s`, on a Windows platform.

```
s = serial('COM1');
set(s,'BaudRate',9600,'Parity','even')
set(s,{'StopBits','RecordName'},{2,'sydney.txt'})
set(s,'Parity')
[ {none} | odd | even | mark | space ]
```

See Also

`get`

Purpose

Configure or display timer object properties

Syntax

```
set(obj)
prop_struct = set(obj)
set(obj,'PropertyName')
prop_cell=set(obj,'PropertyName')
set(obj,'PropertyName',PropertyValue,...)
set(obj,S)
set(obj,PN,PV)
```

Description

`set(obj)` displays property names and their possible values for all configurable properties of timer object `obj`. `obj` must be a single timer object.

`prop_struct = set(obj)` returns the property names and their possible values for all configurable properties of timer object `obj`. `obj` must be a single timer object. The return value, `prop_struct`, is a structure whose field names are the property names of `obj`, and whose values are cell arrays of possible property values or empty cell arrays if the property does not have a finite set of possible string values.

`set(obj,'`*`PropertyName`*`')` displays the possible values for the specified property, *`PropertyName`*, of timer object `obj`. `obj` must be a single timer object.

`prop_cell=set(obj,'`*`PropertyName`*`')` returns the possible values for the specified property, *`PropertyName`*, of timer object `obj`. `obj` must be a single timer object. The returned array, `prop_cell`, is a cell array of possible value strings or an empty cell array if the property does not have a finite set of possible string values.

`set(obj,'`*`PropertyName`*`',PropertyValue,...)` configures the property, *`PropertyName`*, to the specified value, `PropertyValue`, for timer object `obj`. You can specify multiple property name/property value pairs in a single statement. `obj` can be a single timer object or a vector of timer objects, in which case `set` configures the property values for all the timer objects specified.

`set(obj,S)` configures the properties of `obj`, with the values specified in `S`, where `S` is a structure whose field names are object property names.

`set(obj,PN,PV)` configures the properties specified in the cell array of strings, `PN`, to the corresponding values in the cell array `PV`, for the timer object `obj`. `PN` must be a vector. If `obj` is an array of timer objects, `PV` can be an M-by-N cell array, where M is equal to the length of timer object array and N is equal to the length of `PN`. In this case, each timer object is updated with a different set of values for the list of property names contained in `PN`.

Note When specifying parameter/value pairs, you can use any mixture of strings, structures, and cell arrays in the same call to `set`.

Examples

Create a timer object.

```
t = timer;
```

Display all configurable properties and their possible values.

```
set(t)
    BusyMode: [ {drop} | queue | error ]
    ErrorFcn: string -or- function handle -or- cell array
    ExecutionMode: [ {singleShot} | fixedSpacing | fixedDelay | fixedRate ]
    Name
    ObjectVisibility: [ {on} | off ]
    Period
    StartDelay
    StartFcn: string -or- function handle -or- cell array
    StopFcn: string -or- function handle -or- cell array
    Tag
    TasksToExecute
    TimerFcn: string -or- function handle -or- cell array
    UserData
```

View the possible values of the `ExecutionMode` property.

```
set(t, 'ExecutionMode')
[ {singleShot} | fixedSpacing | fixedDelay | fixedRate ]
```

Set the value of a specific timer object property.

```
set(t, 'ExecutionMode', 'FixedRate')
```

Set the values of several properties of the timer object.

```
set(t, 'TimerFcn', 'callbk', 'Period', 10)
```

Use a cell array to specify the names of the properties you want to set and another cell array to specify the values of these properties.

```
set(t, {'StartDelay', 'Period'}, {30, 30})
```

See Also

`timer` | `get(timer)`

set (tscollection)

Purpose

Set properties of `tscollection` object

Syntax

```
set(tsc,'Property',Value)
set(tsc,'Property1',Value1,'Property2',Value2,...)
set(tsc,'Property')
```

Description

`set(tsc,'Property',Value)` sets the property `'Property'` of the `tscollection` `tsc` to the value `Value`. The following syntax is equivalent:

```
tsc.Property = Value
```

`set(tsc,'Property1',Value1,'Property2',Value2,...)` sets multiple property values for `tsc` with a single statement.

`set(tsc,'Property')` displays values for the specified property in the time-series collection `tsc`.

`set(tsc)` displays all properties and values of the `tscollection` object `tsc`.

See Also

`get (tscollection)`

setabstime (tscollection)

Purpose

Set times of `tscollection` object as date strings

Syntax

```
tsc = setabstime(tsc,Times)
tsc = setabstime(tsc,Times,format)
```

Description

`tsc = setabstime(tsc,Times)` sets the times in tsc using the date strings Times. Times must be either a cell array of strings, or a `char` array containing valid date or time values in the same date format.

`tsc = setabstime(tsc,Times,format)` specifies the date-string format used in Times explicitly.

Examples

1 Create a `tscollection` object.

```
tsc = tscollection(timeseries(rand(3,1)))
```

2 Set the absolute time vector.

```
tsc = setabstime(tsc,{'12-DEC-2005 12:34:56',...
      '12-DEC-2005 13:34:56','12-DEC-2005 14:34:56'})
```

See Also

`datestr` | `tscollection` | `getabstime (tscollection)`

setappdata

Purpose

Specify application-defined data

Syntax

```
setappdata(h,'name',value)
```

Description

`setappdata(h,'name',value)` sets application-defined data for the object with handle `h`. The application-defined data, which is created if it does not already exist, is assigned the specified `name` and `value`. The `value` can be any type of data.

Tips

Application data is data that is meaningful to or defined by your application which you attach to a figure or any GUI component (other than ActiveX controls) through its `AppData` property. Only Handle Graphics MATLAB objects use this property.

See Also

`getappdata` | `isappdata` | `rmappdata`

Purpose

Set default random number stream

Note `RandStream.setDefaultStream` will be removed in a future version. Use `RandStream.setGlobalStream` instead. The shared global stream used by `rand`, `randi`, and `randn`, referred to in former versions of MATLAB as the *default stream* is now referred to as the *global stream*.

Syntax

```
prevstream = RandStream.setDefaultStream(stream)
```

Description

`prevstream = RandStream.setDefaultStream(stream)` returns the current global random number stream, and designates the random number stream `stream` as the new random number stream to be used by the `rand`, `randi`, and `randn` functions.

setdiff

Purpose

Set difference of two arrays

Syntax

```
C = setdiff(A,B)
C = setdiff(A,B,'rows')
[C,ia] = setdiff(A,B)
[C,ia] = setdiff(A,B,'rows')

[C,ia] = setdiff(___,setOrder)
[C,ia] = setdiff(A,B,'legacy')
[C,ia] = setdiff(A,B,'rows','legacy')
```

Description

`C = setdiff(A,B)` returns the values in `A` that are not in `B`. The values of `C` are in sorted order.

`C = setdiff(A,B,'rows')` treats each row of `A` and each row of `B` as single entities and returns the rows from `A` that are not in `B`. The rows of matrix `C` are in sorted order.

The `'rows'` option does not support cell arrays.

`[C,ia] = setdiff(A,B)` also returns the index vector `ia`, such that `C = A(ia)`.

`[C,ia] = setdiff(A,B,'rows')` also returns the index vector `ia`, such that `C = A(ia,:)`.

`[C,ia] = setdiff(___,setOrder)` returns `C` in a specific order using any of the input arguments in the previous syntaxes. `setOrder='sorted'` returns the values (or rows) of `C` in sorted order. `setOrder='stable'` returns the values (or rows) of `C` in the same order as `A`. If no value is specified, the default is `'sorted'`.

`[C,ia] = setdiff(A,B,'legacy')` and `[C,ia] = setdiff(A,B,'rows','legacy')` preserve the behavior of the `setdiff` function from R2012b and prior releases.

Input Arguments

A,B - Input arrays

vectors | matrices | N-D arrays

Input arrays, specified as vectors, matrices, or N-D arrays whose elements can be any numeric, `logical`, or `char` data type. `A` and `B` also can be cell arrays of strings. Furthermore, `A` and `B` can be objects with the following class methods:

- `sort` (or `sortrows` for the `'rows'` option), where the second output of this method is `double` or another built-in numeric class
- `eq`
- `ne`

These objects include heterogeneous arrays derived from the same root class.

`A` and `B` must belong to the same class with the following exceptions:

- `logical`, `char`, and all numeric classes can combine with `double` arrays.
- Cell arrays of strings can combine with `char` arrays.

If you specify the `'rows'` option, `A` and `B` must have the same number of columns.

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

setOrder - Order flag

`'sorted'` (default) | `'stable'`

Order flag, specified as `'sorted'` or `'stable'`, indicates the order of the values (or rows) in `C`.

Order Flag	Meaning
`'sorted'`	The values (or rows) in `C` return in sorted order. For example: `C = setdiff([4 1 3 2],[2 1],'sorted')` returns `C = [3 4]`.
`'stable'`	The values (or rows) in `C` return in the same order as in `A`. For example: `C = setdiff([4 1 3 2],[2 1],'stable')` returns `C = [4 3]`.

Data Types
`char`

Output Arguments

`C` - Difference of `A` and `B`

Vector | Matrix

Difference of `A` and `B`, returned as a vector or matrix. The following describes the shape of `C` when the `'legacy'` flag is not specified:

- If the `'rows'` flag is not specified and `A` is a row vector, then `C` is a row vector.
- If the `'rows'` flag is not specified and `A` is not a row vector, then `C` is a column vector.
- If the `'rows'` flag is specified, then `C` is a matrix containing the rows of `A` that are not in `B`.
- If all the values (or rows) of `A` are also in `B`, then `C` is an empty matrix.

The class of `C` is the same as the class of `A`, unless `A` is a character array and `B` is a cell array of strings, in which case `C` is a cell array of strings.

`ia` - Index to `A`

column vector

Index to `A`, returned as a column vector when the `'legacy'` flag is not specified. `ia` identifies the values (or rows) in `A` that are not in `B`. If

there is a repeated value (or row) appearing exclusively in `A`, then `ia` contains the index to the first occurrence of the value (or row).

Examples

Difference of Two Vectors

Define two vectors with values in common.

```
A = [3 6 2 1 5 1 1]; B = [2 4 6];
```

Find the values in `A` that are not in `B`.

```
C = setdiff(A,B)
```

```
C =

     1     3     5
```

Difference of Two Vectors and the Indices to the Different Values

Define two vectors with values in common.

```
A = [3 6 2 1 5 1 1]; B = [2 4 6];
```

Find the values in `A` that are not in `B` as well as the index vector `ia`, such that `C = A(ia)`.

```
[C,ia] = setdiff(A,B)
```

```
C =

     1     3     5

ia =

     4
     1
     5
```

Difference of the Rows in Two Matrices

Define two matrices with rows in common.

```
A = [7 9 7; 0 0 0; 7 9 7; 5 5 5; 1 4 5];
B = [0 0 0; 5 5 5];
```

Find the rows from `A` that are not in `B` as well as the index vector `ia`, such that `C = A(ia,:)`.

```
[C,ia] = setdiff(A,B,'rows')

C =

     1     4     5
     7     9     7

ia =

     5
     1
```

Difference of Two Vectors with Specified Output Order

Use the `setOrder` argument to specify the ordering of the values in `C`.

Specify `'stable'` or `'sorted'` when the order of the values in `C` are important.

```
A = [3 6 2 1 5 1 1]; B = [2 4 6];
[C,ia] = setdiff(A,B,'stable')

C =

     3     1     5

ia =

     1
```

```
     4
     5
```

Alternatively, you can specify `'sorted'` order.

```
 [C,ia] = setdiff(A,B,'sorted')
```

```
C =

     1     3     5

ia =

     4
     1
     5
```

Difference of Vectors Containing NaNs

Define two vectors containing `NaN`.

```
A = [5 NaN NaN]; B = [5 NaN];
```

Find the set difference of `A` and `B`.

```
C = setdiff(A,B)
```

```
C =

   NaN   NaN
```

`setdiff` treats `NaN` values as distinct.

Cell Array of Strings with Trailing White Space

Create a cell array of strings, `A`.

```
A = {'dog','cat','fish','horse'};
```

Create a cell array of strings, B, where some of the strings have trailing white space.

```
B = {'dog ','cat','fish ','horse'};
```

Find the strings in A that are not in B.

```
[C,ia] = setdiff(A,B)

C =

    'dog'    'fish'


ia =

     1
     3
```

setdiff treats trailing white space in cell arrays of strings as distinct characters.

Difference of Char and Cell Array of Strings

Create a character array, A.

```
A = ['cat';'dog';'fox';'pig'];
class(A)

ans =

char
```

Create a cell array of strings, B.

```
B={'dog','cat','fish','horse'};
class(B)

ans =
```

```
cell
```

Find the strings in A that are not in B.

```
C = setdiff(A,B)
```

```
C =

    'fox'
    'pig'
```

The result, C, is a cell array of strings.

```
class(C)
```

```
ans =

cell
```

Preserve Legacy Behavior of setdiff

Use the 'legacy' flag to preserve the behavior of setdiff from R2012b and prior releases in your code.

Find the difference of A and B with the current behavior.

```
A = [3 6 2 1 5 1 1]; B = [2 4 6];
[C1,ia1] = setdiff(A,B)
```

```
C1 =

     1     3     5

ia1 =

     4
     1
     5
```

Find the difference of A and B, and preserve the legacy behavior.

```
[C2,ia2] = setdiff(A,B,'legacy')

C2 =

     1     3     5

ia2 =

     7     1     5
```

See Also

unique | intersect | ismember | issorted | union | setxor | sort

Purpose

Make specified IFD current IFD

Syntax

`tiffobj.setDirectory(dirNum)`

Description

`tiffobj.setDirectory(dirNum)` sets the image file directory (IFD) specified by `dirNum` as the current IFD. `Tiff` object methods operate on the current IFD. The directory index number is one-based.

Examples

Open a TIFF file and move to an IFD in the file by specifying its index number. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path. The TIFF file should contain multiple images.

```
t = Tiff('myfile.tif','r');
t.setDirectory(2);
```

References

This method corresponds to the `TIFFSetDirectory` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff.currentDirectory` | `Tiff.nextDirectory`

Tutorials

- “Exporting Image Data and Metadata to TIFF Files”
- “Reading Image Data and Metadata from TIFF Files”

setdisp (hgsetget)

Purpose

Override to change command window display

Syntax

```
setdisp(H)
```

Description

`setdisp(H)` called by `set` when `set` is called with no output arguments and a single input argument that is a handle array. Override this `hgsetget` class method in a subclass to change how property information is displayed in the command window.

See Also

`hgsetget | set (hgsetget)`

How To

- "Implementing a Set/Get Interface for Properties"

Purpose

Set environment variable

Syntax

```
setenv(name,value)
setenv(name)
```

Description

`setenv(name,value)` sets the value of an environment variable belonging to the underlying operating system. Inputs `name` and `value` are both strings. If `name` already exists as an environment variable, then `setenv` replaces its current value with the string given in `value`. If `name` does not exist, `setenv` creates a new environment variable called `name` and assigns `value` to it.

`setenv(name)` is equivalent to `setenv(name, '')` and assigns a null value to the variable `name`. On the Microsoft Windows platform, this is equivalent to undefining the variable. On most UNIX platforms, it is possible to have an environment variable defined as empty.

The maximum number of characters in `name` is 2^{15} - 2 (or 32766). If `name` contains the character =, `setenv` throws an error. The behavior of environment variables with = in the name is not well-defined.

On all platforms, `setenv` passes the `name` and `value` strings to the operating system unchanged. Special characters such as `;`, `/`, `:`, `$`, `%`, etc. are left unexpanded and intact in the variable value.

Values assigned to variables using `setenv` are picked up by any process that is spawned using the MATLAB `system`, `unix`, `dos` or `!` functions. You can retrieve any value set with `setenv` by using `getenv(name)`.

Examples

% Set and retrieve a new value for the environment variable `TEMP`:

```
setenv('TEMP', 'C:\TEMP');
getenv('TEMP')
```

% Append the `Perl\bin` folder to your system `PATH` variable:

```
setenv('PATH', [getenv('PATH') ';D:\Perl\bin']);
```

See Also

`getenv` | `system` | `unix` | `dos` | `!`

setfield

Purpose

Assign values to structure array field

Syntax

```
s = setfield(s,'field',value)
s = setfield(s,{sIndx1,...,sIndxM},'field',{fIndx1,...,
   fIndxN},value)
```

Description

`s = setfield(s,'field',value)`, where `s` is a 1-by-1 structure, sets the contents of the specified field, equivalent to `s.field = value`. If `s` does not contain the specified *field*, the `setfield` function creates the field and assigns the specified value. Pass field references as strings.

`s = setfield(s,{sIndx1,...,sIndxM},'field',{fIndx1,...,fIndxN},value)` sets the contents of the specified field, equivalent to `s(sIndx1,...,sIndxM).field(fIndx1,...,fIndxN) = value`. The `setfield` function supports multiple sets of *field* and *fIndx* inputs. If structure `s` or any of the fields is a nonscalar structure, the *Indx* inputs associated with that input are required. Otherwise, the *Indx* inputs are optional. If you specify a single colon operator for an index input, enclose it in single quotation marks: `':'`.

Tips

- For most cases, add data to a structure array by indexing rather than using the `setfield` function. For more information, see “Access Data in a Structure Array” and “Generate Field Names from Variables”.
- Call `setfield` to simplify references to structure arrays with nested fields, as shown in the Examples section.

Examples

Add values to a structure that contains nested fields:

```
grades = [];
level = 5;
semester = 'Fall';
subject = 'Math';
student = 'John_Doe';
fieldnames = {semester subject student}
newGrades_Doe = [85, 89, 76, 93, 85, 91, 68, 84, 95, 73];
```

```
grades = setfield(grades, {level}, ...
                  fieldnames{:}, {10, 21:30}, ...
                  newGrades_Doe);

% View the new contents.
grades(level).(semester).(subject).(student)(10, 21:30)
```

Using the structure defined in the previous example, remove the tenth row of the specified field:

```
grades = setfield(grades, {level}, fieldnames{:}, {10,':'}, []);
```

See Also

getfield | fieldnames | isfield | orderfields | rmfield

How To

- "Generate Field Names from Variables"
- "Access Data in a Structure Array"

setpixelposition

Purpose

Set component position in pixels

Syntax

```
setpixelposition(handle,position)
setpixelposition(handle,position,recursive)
```

Description

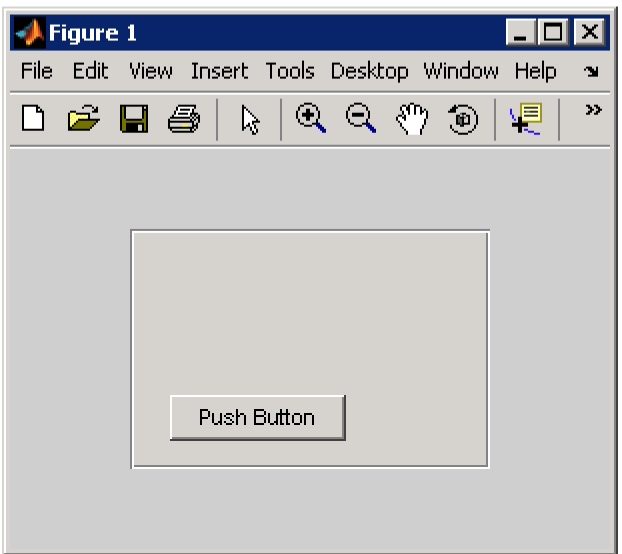
`setpixelposition(handle,position)` sets the position of the component specified by `handle`, to the specified position relative to its parent. `position` is a four-element vector that specifies the location and size of the component: [pixels from left, pixels from bottom, pixels across, pixels high].

`setpixelposition(handle,position,recursive)` sets the position as above. If Boolean `recursive` is true, the position is set relative to the parent figure of `handle`.

Examples

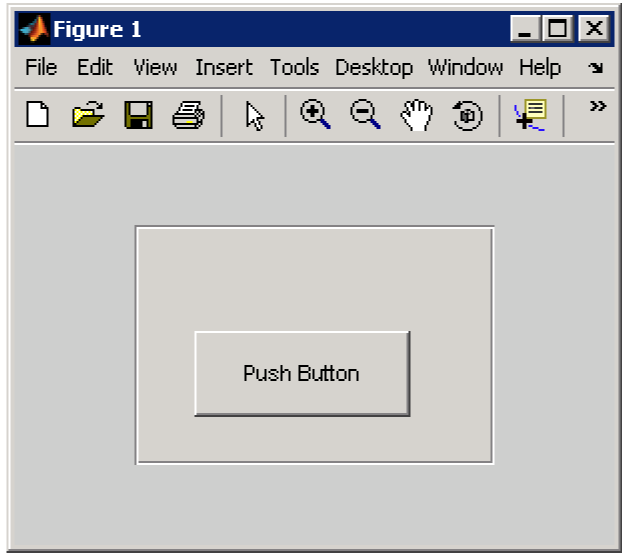
This example first creates a push button within a panel.

```
f = figure('Position',[300 300 300 200]);
p = uipanel('Position',[.2 .2 .6 .6]);
h1 = uicontrol(p,'Style','PushButton','Units','normalized',...
               'String','Push Button','Position',[.1 .1 .5 .2]);
```



The example then retrieves the position of the push button and changes its position with respect to the panel.

```
pos1 = getpixelposition(h1);
setpixelposition(h1,pos1 + [10 10 25 25]);
```



See Also

getpixelposition | uicontrol | uipanel

Purpose Set preference

Syntax

```
setpref('group','pref',val)
setpref('group',{'pref1','pref2',...,'prefn'},{val1,val2,...,
   valn})
```

Description

`setpref('group','pref',val)` sets the preference specified by `group` and `pref` to the value `val`. Individual preference values can be any MATLAB data type, including numeric types, strings, cell arrays, structures, and objects. Setting a preference that does not yet exist causes it to be created.

`group` labels a related collection of preferences. You can choose any name that is a legal variable name, and is descriptive enough to be unique, e.g., `'MathWorks_GUIDE_ApplicationPrefs'`. The input argument `pref` identifies an individual preference in that group, and must be a legal variable name.

`setpref('group',{'pref1','pref2',...,'prefn'},{val1,val2,...,valn})` sets each preference specified in the cell array of names to the corresponding value.

Note Preference values are persistent and maintain their values between MATLAB sessions. Where they are stored is system dependent.

Examples

Use `addpref` to create a preference group called `mytoolbox` and a preference within it called `version`, and then modify the contents of `version` using `setpref`:

```
addpref('mytoolbox','version','1.0')
getpref('mytoolbox','version')

ans =
1.0

setpref('mytoolbox','version',{'1.0','beta'})
```

```
getpref('mytoolbox','version')

ans =
    '1.0'    'beta'
```

See Also

addpref | getpref | ispref | rmpref | uigetpref | uisetpref

Purpose

Set string flag

Note `setstr` is not recommended. Use `char` instead.

Description

This MATLAB 4 function has been renamed `char` in MATLAB 5.

Tiff.setSubDirectory

Purpose

Make subIFD specified by byte offset current IFD

Syntax

`tiffobj.setSubDirectory(offset)`

Description

`tiffobj.setSubDirectory(offset)` sets the subimage file directory (subIFD) specified by `offset` the current IFD. The offset value is given in bytes. Use this method when you want to access subIFDs linked through the `SubIFD` tag.

Examples

Open a TIFF file and read the value of the `SubIFD` tag in the current IFD. The `SubIFD` tag contains byte offsets that specify the location of subIFDs in the IFD. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path. The TIFF file should contain subIFDs.

```
t = Tiff('myfile.tif','r');
%
% Read the value of the SubIFD tag to get subdirectory offsets.
offsets = t.getTag('SubIFD');
%
% Set one of the subdirectories (if more than one) as the current directo
t.setSubDirectory(offsets(1));
```

References

This method corresponds to the `TIFFSetSubDirectory` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff.setDirectory`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

Purpose

Set value of tag

Syntax

```
tiffobj.setTag(tagId,tagValue)
tiffobj.setTag(tagStruct)
```

Description

`tiffobj.setTag(tagId,tagValue)` sets the value of the TIFF tag specified by `tagId` to the value specified by `tagValue`. You can specify `tagId` as a character string (`'ImageWidth'`) or using the numeric tag identifier defined by the TIFF specification (256). To see a list of all the tags with their numeric identifiers, view the value of the Tiff object `TagID` property. Use the `TagID` property to specify the value of a tag. For example, `Tiff.TagID.ImageWidth` is equivalent to the tag's numeric identifier.

`tiffobj.setTag(tagStruct)` sets the values of all of the tags with name/value fields in `tagStruct`. The names of fields in `tagstruct` must be the name of TIFF tags.

Note If you are modifying a tag rather than creating it, you must use the `Tiff.rewriteDirectory` method after using the `Tiff.setTag` method.

Examples

Create a structure with fields named after TIFF tags and assign values to the fields. Pass this structure to the `setTag` method to set the values of these tags. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path.

```
t = Tiff('myfile.tif', 'r+');

tagStruct.ImageWidth = 1600;
tagStruct.ImageLength = 3200;
tagStruct.Photometric = Tiff.Photometric.RGB;
tagStruct.BitsPerSample = 8;
tagStruct.SamplesPerPixel = 3;
tagStruct.TileWidth = 160;
```

```
tagStruct.TileLength = 320;
tagStruct.PlanarConfiguration = Tiff.PlanarConfiguration.Chunky;
tagStruct.Software = 'MATLAB';
t.setTag(tagStruct);
```

References

This method corresponds to the `TIFFSetField` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff.getTag`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

Purpose Change name of `timeseries` object in `tscollection`

Syntax

```
tsc = settimeseriesnames(tsc,old,new)
```

Description `tsc = settimeseriesnames(tsc,old,new)` replaces the `old` name of `timeseries` object with the new name in tsc.

See Also `tscollection`

setxor

Purpose

Set exclusive OR of two arrays

Syntax

```
C = setxor(A,B)
C = setxor(A,B,'rows')
[C,ia,ib] = setxor(A,B)
[C,ia,ib] = setxor(A,B,'rows')

[C,ia,ib] = setxor( ___ ,setOrder)

[C,ia,ib] = setxor(A,B,'legacy')

[C,ia,ib] = setxor(A,B,'rows','legacy')
```

Description

`C = setxor(A,B)` returns the values of `A` and `B` that are not in their intersection (the symmetric difference). The values of `C` are in sorted order.

`C = setxor(A,B,'rows')` treats each row of `A` and each row of `B` as single entities and returns the rows of matrices `A` and `B` that are not in their intersection. The rows of the matrix `C` are in sorted order.

The `'rows'` option does not support cell arrays.

`[C,ia,ib] = setxor(A,B)` also returns index vectors `ia` and `ib`, such that `C` is a sorted combination of the elements `A(ia)` and `B(ib)`.

`[C,ia,ib] = setxor(A,B,'rows')` also returns index vectors `ia` and `ib`, such that `C` is a sorted combination of the rows of `A(ia,:)` and `B(ib,:)`.

`[C,ia,ib] = setxor( ___ ,setOrder)` returns `C` in a specific order using any of the input arguments in the previous syntaxes. `setOrder='sorted'` returns the values (or rows) of `C` in sorted order. `setOrder='stable'` returns the values (or rows) of `C` in the same order as `A` and `B`. If no value is specified, the default is `'sorted'`.

`[C,ia,ib] = setxor(A,B,'legacy')` and `[C,ia,ib] = setxor(A,B,'rows','legacy')` preserve the behavior of the `setxor` function from R2012b and prior releases.

Input Arguments

A,B - Input arrays

vectors | matrices | N-D arrays

Input arrays, specified as vectors, matrices, or N-D arrays whose elements can be any numeric, `logical`, or `char` data type. `A` and `B` also can be cell arrays of strings. Furthermore, `A` and `B` can be objects with the following class methods:

- `sort` (or `sortrows` for the `'rows'` option), where the second output of this method is `double` or another built-in numeric class
- `eq`
- `ne`

These objects include heterogeneous arrays derived from the same root class.

`A` and `B` must belong to the same class with the following exceptions:

- `logical`, `char`, and all numeric classes can combine with `double` arrays.
- Cell arrays of strings can combine with `char` arrays.

If you specify the `'rows'` option, `A` and `B` must have the same number of columns.

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

setOrder - Order flag

`'sorted'` (default) | `'stable'`

Order flag, specified as `'sorted'` or `'stable'`, indicates the order of the values (or rows) in `C`.

Order Flag	Meaning
'sorted'	The values (or rows) in C return in sorted order. For example: `C = setxor([5 1 3],[4 1 2],'sorted')` returns `C = [2 3 4 5]`.
'stable'	The values (or rows) in C return in the same order as in A and B. For example: `C = setxor([5 1 3],[4 1 2],'stable')` returns `C = [5 3 4 2]`.

Data Types
char

Output Arguments

C - Symmetric difference array

vector | matrix

Symmetric difference array, returned as a vector or matrix. The following describes the shape of C when the 'legacy' flag is not specified:

- If the 'rows' flag is not specified, then C is a column vector unless both A and B are row vectors.
- If the 'rows' flag is not specified and both A and B are row vectors, then C is a row vector.
- If the 'rows' flag is specified, then C is a matrix containing the rows of A and B that are not in the intersection.
- If all the values (or rows) of A are also in B, then C is an empty matrix.

The class of the inputs A and B determines the class of C:

- If the class of A and B are the same, then C is the same class.
- If you combine a char or nondouble numeric class with double, then C is the same class as the nondouble input.
- If you combine a logical class with double, then C is double.

- If you combine a cell array of strings with `char`, then `C` is a cell array of strings.

ia - Index to `A`
column vector

Index to `A`, returned as a column vector when the `'legacy'` flag is not specified. `ia` identifies the values (or rows) in `A` that contribute to the symmetric difference. If there is a repeated value (or row) appearing exclusively in `A`, then `ia` contains the index to the first occurrence of the value (or row).

ib - Index to `B`
column vector

Index to `B`, returned as a column vector when the `'legacy'` flag is not specified. `ib` identifies the values (or rows) in `B` that contribute to the symmetric difference. If there is a repeated value (or row) appearing exclusively in `B`, then `ib` contains the index to the first occurrence of the value (or row).

Examples

Symmetric Difference of Two Vectors

Define two vectors with a value in common.

```
A = [5 1 3 3 3]; B = [4 1 2];
```

Find the values of `A` and `B` that are not in their intersection.

```
C = setxor(A,B)
```

```
C =

     2     3     4     5
```

Symmetric Difference of Two Vectors and the Indices to the Different Values

Define two vectors with a value in common.

```
A = [5 1 3 3 3]; B = [4 1 2];
```

Find the values of `A` and `B` that are not in their intersection as well as the index vectors `ia` and `ib`.

```
[C,ia,ib] = setxor(A,B)

C =

     2     3     4     5


ia =

     3
     1


ib =

     3
     1
```

`C` is a sorted combination of the elements `A(ia)` and `B(ib)`.

Symmetric Difference of the Rows in Two Matrices

Define two matrices with rows in common.

```
A = [7 8 9; 7 7 1; 7 7 1; 1 2 3; 4 5 6];
B = [1 2 3; 4 5 6; 7 7 2];
```

Find the rows of `A` and `B` that are not in their intersection as well as the index vectors `ia` and `ib`.

```
[C,ia,ib] = setxor(A,B,'rows')

C =
```

```
     7     7     1
     7     7     2
     7     8     9


ia =

     2
     1


ib =

     3
```

`C` is a sorted combination of the rows of `A(ia,:)` and `B(ib,:)`.

Symmetric Difference of Two Vectors in a Specified Order

Use the `setOrder` argument to specify the ordering of the values in `C`.

Specify `'stable'` if you want the values in `C` to have the same order as `A` and `B`.

```
A = [5 1 3 3 3]; B = [4 1 2];
[C,ia,ib] = setxor(A,B,'stable')
```

```
C =

     5     3     4     2


ia =

     1
     3


ib =
```

```
     1
     3
```

Alternatively, you can specify `'sorted'` order.

```
[C,ia,ib] = setxor(A,B,'sorted')
```

```
C =

     2     3     4     5


ia =

     3
     1


ib =

     3
     1
```

Symmetric Difference of Vectors Containing NaNs

Define two vectors containing `NaN`.

```
A = [5 NaN NaN]; B = [5 NaN NaN];
```

Find the symmetric difference of vectors `A` and `B`.

```
C = setxor(A,B)
```

```
C =

   NaN   NaN   NaN   NaN
```

The `setxor` function treats `NaN` values as distinct.

Cell Array of Strings with Trailing White Space

Create a cell array of strings, A.

```
A = {'dog','cat','fish','horse'};
```

Create a cell array of strings, B, where some of the strings have trailing white space.

```
B = {'dog ','cat','fish ','horse'};
```

Find the strings that are not in the intersection of A and B.

```
[C,ia,ib] = setxor(A,B)

C =

    'dog'    'dog '    'fish'    'fish '


ia =

     1
     3


ib =

     1
     3
```

setxor treats trailing white space in cell arrays of strings as distinct characters.

Symmetric Difference of Vectors of Different Classes and Shapes

Create a column vector character array.

```
A = ['A';'B';'C'], class(A)
```

```
A =

A
B
C


ans =

char
```

Create a row vector containing elements of numeric type `double`.

```
B = [66 67 68], class(B)
```

```
B =

    66    67    68


ans =

double
```

Find the symmetric difference of `A` and `B`.

```
C = setxor(A,B)
```

```
C =

A
D
```

The result is a column vector character array.

```
class(C)
```

```
ans =

char
```

Symmetric Difference of Char and Cell Array of Strings

Create a character array, `A`.

```
A = ['cat';'dog';'fox';'pig'];
class(A)
```

```
ans =

char
```

Create a cell array of strings, `B`.

```
B={'dog','cat','fish','horse'};
class(B)
```

```
ans =

cell
```

Find the strings that are not in the intersection of `A` and `B`.

```
C = setxor(A,B)
```

```
C =

    'fish'
    'fox'
    'horse'
    'pig'
```

The result, `C`, is a cell array of strings.

```
class(C)
```

```
ans =
```

```
cell
```

Preserve Legacy Behavior of setxor

Use the `'legacy'` flag to preserve the behavior of `setxor` from R2012b and prior releases in your code.

Find the symmetric difference of `A` and `B` with the current behavior.

```
A = [5 1 3 3 3]; B = [4 1 2 2];
[C1,ia1,ib1] = setxor(A,B)

C1 =

     2     3     4     5


ia1 =

     3
     1


ib1 =

     3
     1
```

Find the symmetric difference and preserve the legacy behavior.

```
[C2,ia2,ib2] = setxor(A,B,'legacy')

C2 =

     2     3     4     5


ia2 =
```

```
     5     1


ib2 =

     4     1
```

See Also

`unique` | `intersect` | `ismember` | `issorted` | `setdiff` | `union` | `sort`

shading

Purpose

Set color shading properties

Syntax

```
shading flat
shading faceted
shading interp
shading(axes_handle,...)
```

Description

The `shading` function controls the color shading of surface and patch graphics objects.

`shading flat` each mesh line segment and face has a constant color determined by the color value at the endpoint of the segment or the corner of the face that has the smallest index or indices.

`shading faceted` flat shading with superimposed black mesh lines. This is the default shading mode.

`shading interp` varies the color in each line segment and face by interpolating the colormap index or true color value across the line or face.

`shading(axes_handle,...)` applies the shading type to the objects in the axes specified by `axes_handle`, instead of the current axes. Use quoted strings when using a function form. For example:

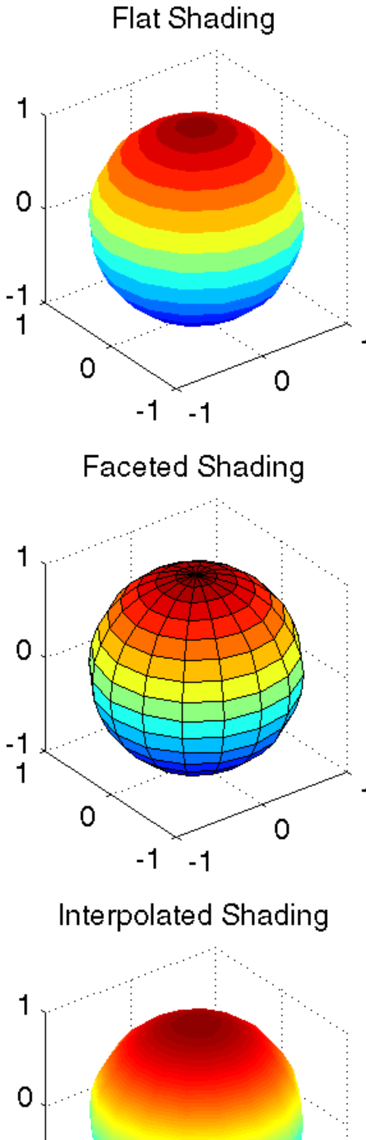
```
shading(gca,'interp')
```

Examples

Compare a flat, faceted, and interpolated-shaded sphere.

```
subplot(3,1,1)
sphere(16)
axis square
shading flat
title('Flat Shading')

subplot(3,1,2)
sphere(16)
axis square
shading faceted
```

```
title('Faceted Shading')

subplot(3,1,3)
sphere(16)
axis square
shading interp
title('Interpolated Shading')
```


Flat Shading
1
0
-1
1
0
-1
-1
0
1
Faceted Shading
1
0
-1
1
0
-1
-1
0
1
Interpolated Shading
1
0
-1
1
0
-1
-1
0
1

Algorithms

`shading` sets the `EdgeColor` and `FaceColor` properties of all surface and patch graphics objects in the current axes. `shading` sets the appropriate values, depending on whether the surface or patch objects represent meshes or solid surfaces.

See Also

`fill` | `fill3` | `hidden` | `light` | `lighting` | `mesh` | `patch` | `pcolor` | `surf` | patch | surface

shg

Purpose Show most recent graph window

Syntax `shg`

Description `shg` makes the current figure visible and raises it above all other figures on the screen. This is identical to using the command `figure(gcf)`.

See Also `figure` | `gca` | `gcf`

Purpose

Shift dimensions

Syntax

```
B = shiftdim(X,n)
[B,nshifts] = shiftdim(X)
```

Description

`B = shiftdim(X,n)` shifts the dimensions of `X` by `n`. When `n` is positive, `shiftdim` shifts the dimensions to the left and wraps the `n` leading dimensions to the end. When `n` is negative, `shiftdim` shifts the dimensions to the right and pads with singletons.

`[B,nshifts] = shiftdim(X)` returns the array `B` with the same number of elements as `X` but with any leading singleton dimensions removed. A singleton dimension is any dimension for which `size(A,dim) = 1`. `nshifts` is the number of dimensions that are removed.

If `X` is a scalar, `shiftdim` has no effect.

Examples

The `shiftdim` command is handy for creating functions that, like `sum` or `diff`, work along the first nonsingleton dimension.

```
a = rand(1,1,3,1,2);
[b,n]  = shiftdim(a); % b is 3-by-1-by-2 and n is 2.
c = shiftdim(b,-n);   % c == a.
d = shiftdim(a,3);    % d is 1-by-2-by-1-by-1-by-3.
```

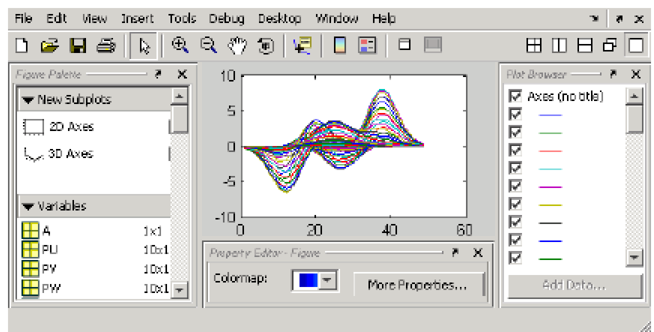
See Also

`circshift` | `reshape` | `squeeze` | `permute` | `ipermute`

showplottool

Purpose

Show or hide figure plot tool



Syntax

```
showplottool('tool')
showplottool('on','tool')
showplottool('off','tool')
showplottool('toggle','tool')
showplottool(figure_handle,...)
```

Description

`showplottool('tool')` shows the specified plot tool on the current figure. *`tool`* can be one of the following strings:

- `figurepalette`
- `plotbrowser`
- `propertyeditor`

`showplottool('on','tool')` shows the specified plot tool on the current figure.

`showplottool('off','tool')` hides the specified plot tool on the current figure.

`showplottool('toggle','tool')` toggles the visibility of the specified plot tool on the current figure.

`showplottool(figure_handle,...)` operates on the specified figure instead of the current figure.

Note When you dock, undock, resize, or reposition a plotting tool and then close it, it will still be configured as you left it the next time you open it. There is no command to reset plotting tools to their original, default locations.

Alternatives

Click the larger Plotting Tools icon on the figure toolbar to collectively enable plotting tools, and the smaller icon to collectively disable them. Individually select the **Figure Palette**, **Plot Browser**, and **Property Editor** tools from the figure's **View** menu. For details, see "Plotting Tools — Interactive Plotting".

See Also

`figurepalette` | `plotbrowser` | `plottools` | `propertyeditor`

shrinkfaces

Purpose

Reduce size of patch faces

Syntax

```
shrinkfaces(p,sf)
nfv = shrinkfaces(p,sf)
nfv = shrinkfaces(fv,sf)
shrinkfaces(p)
nfv = shrinkfaces(f,v,sf)
[nf,nv] = shrinkfaces(...)
```

Description

`shrinkfaces(p,sf)` shrinks the area of the faces in patch `p` to shrink factor `sf`. A shrink factor of 0.6 shrinks each face to 60% of its original area. If the patch contains shared vertices, the MATLAB software creates nonshared vertices before performing the face-area reduction.

`nfv = shrinkfaces(p,sf)` returns the face and vertex data in the struct `nfv`, but does not set the `Faces` and `Vertices` properties of patch `p`.

`nfv = shrinkfaces(fv,sf)` uses the face and vertex data from the struct `fv`.

`shrinkfaces(p)` and `shrinkfaces(fv)` (without specifying a shrink factor) assume a shrink factor of 0.3.

`nfv = shrinkfaces(f,v,sf)` uses the face and vertex data from the arrays `f` and `v`.

`[nf,nv] = shrinkfaces(...)` returns the face and vertex data in two separate arrays instead of a struct.

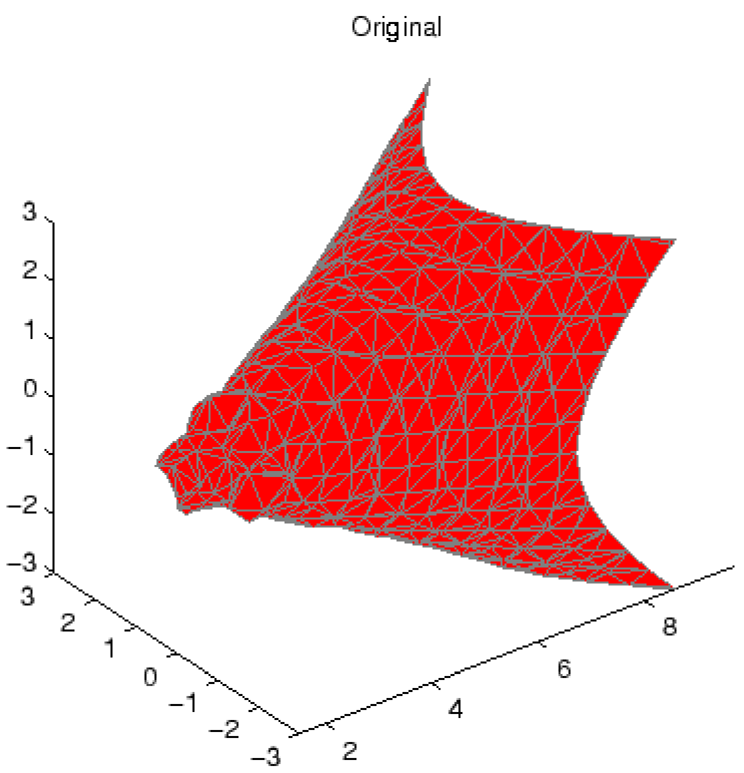
Examples

This example uses the flow data set, which represents the speed profile of a submerged jet within an infinite tank (type `help flow` for more information). Two isosurfaces provide a before and after view of the effects of shrinking the face size.

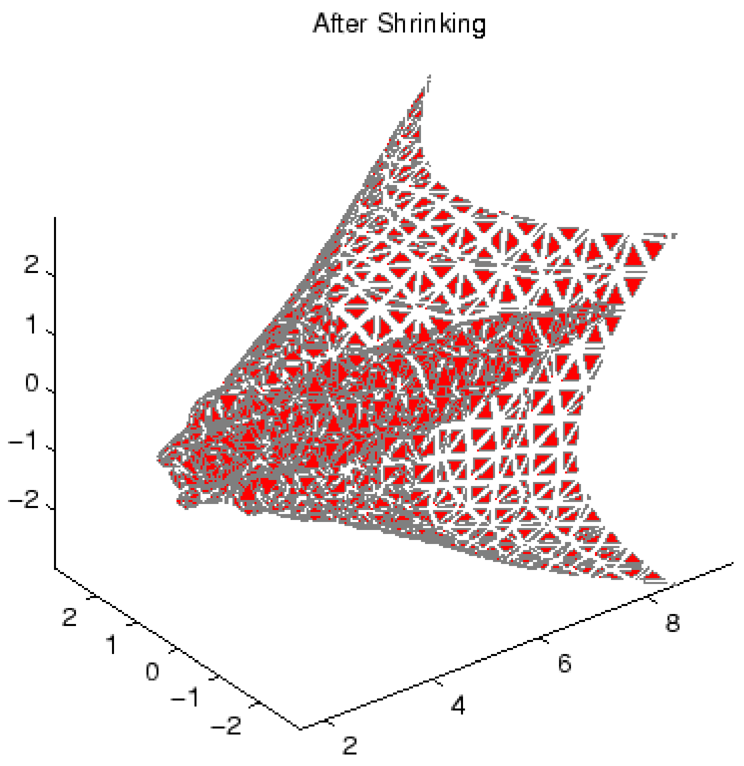
- First `reducevolume` samples the flow data at every other point and then `isosurface` generates the faces and vertices data.
- The `patch` command accepts the face/vertex struct and draws the first (`p1`) isosurface.

- Use the daspect, view, and axis commands to set up the view and then add a title.
- The shrinkfaces command modifies the face/vertex data and passes it directly to patch.

```
[x,y,z,v] = flow;
[x,y,z,v] = reducevolume(x,y,z,v,2);
fv = isosurface(x,y,z,v,-3);
p1 = patch(fv);
set(p1,'FaceColor','red','EdgeColor',[.5,.5,.5]);
daspect([1 1 1]); view(3); axis tight
title('Original')

figure
p2 = patch(shrinkfaces(fv,.3));
set(p2,'FaceColor','red','EdgeColor',[.5,.5,.5]);
daspect([1 1 1]); view(3); axis tight
title('After Shrinking')
```



Original
3
2
1
0
−1
−2
−3
3
2
1
0
−1
−2
−3
2
4
6
8



See Also

`isosurface` | `patch` | `reducevolume` | `daspect` | `view` | `axis`

sign

Purpose

Signum function

Syntax

```
Y = sign(X)
```

Description

`Y = sign(X)` returns an array `Y` the same size as `X`, where each element of `Y` is:

- `1` if the corresponding element of `X` is greater than zero
- `0` if the corresponding element of `X` equals zero
- `-1` if the corresponding element of `X` is less than zero

For nonzero complex `X`, `sign(X) = X./abs(X)`.

See Also

`abs` | `conj` | `imag` | `real`

Purpose

Sine of argument in radians

Syntax

```
Y = sin(X)
```

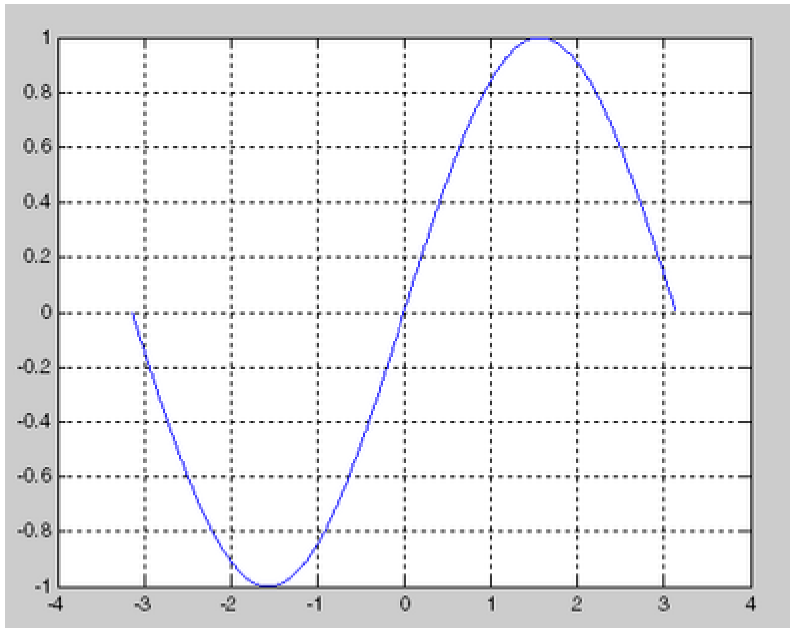
Description

`Y = sin(X)` returns the circular sine of the elements of `X`. The `sin` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

Examples

Graph the sine function over the domain $-\pi \le x \le \pi$.

```
x = -pi:0.01:pi;
plot(x,sin(x)), grid on
```



See Also

`sind` | `asin` | `asind` | `sinh`

sind

Purpose

Sine of argument in degrees

Syntax

```
Y = sind(X)
```

Description

`Y = sind(X)` returns the sine of the elements in `X`, which are expressed in degrees.

Input Arguments

X - Angle in degrees

scalar value | vector | matrix | N-D array

Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `sind` operation is element-wise when `X` is nonscalar.

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

Output Arguments

Y - Sine of angle

scalar value | vector | matrix | N-D array

Sine of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as `X`.

Examples

Sine of 180 degrees compared to sine of radians

```
sind(180)

ans =

     0

sin(pi)

ans =

   1.2246e-16
```

Sine of vector of complex angles, specified in degrees

```
z = [90+i 15+2i 10+3i];
y = sind(z)
```

```
y =

   1.0002          0.2590 + 0.0337i   0.1739 + 0.0516i
```

See Also

sin | asind | asin

Purpose

Convert to single precision

Syntax

```
B = single(A)
```

Description

`B = single(A)` converts the matrix `A` to single precision, returning that value in `B`. `A` can be any numeric object (such as a `double`). If `A` is already single precision, `single` has no effect. Single-precision quantities require less storage than double-precision quantities, but have less precision and a smaller range.

The `single` class is primarily meant to be used to store single-precision values. Hence most operations that manipulate arrays without changing their elements are defined. Examples are `reshape`, `size`, the relational operators, subscripted assignment, and subscripted reference.

You can define your own methods for the `single` class by placing the appropriately named method in an `@single` folder within a folder on your path.

Examples

```
a = magic(4);
b = single(a);

whos
  Name      Size          Bytes  Class

  a         4x4             128  double array
  b         4x4              64  single array
```

See Also

`double`

sinh

Purpose

Hyperbolic sine of argument in radians

Syntax

```
Y = sinh(X)
```

Description

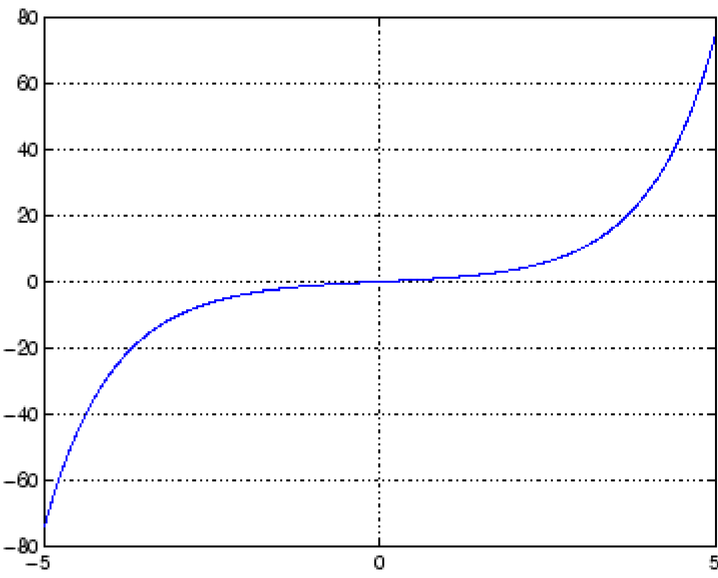
The `sinh` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = sinh(X)` returns the hyperbolic sine of the elements of `X`.

Examples

Graph the hyperbolic sine function over the domain $-5 \le x \le 5$.

```
x = -5:0.01:5;
plot(x,sinh(x)), grid on
```



See Also

`sin` | `asinh` | `cosh`

size

Purpose Array dimensions

Syntax

```
d = size(X)
[m,n] = size(X)
m = size(X,dim)
[d1,d2,d3,...,dn] = size(X),
```

Description

`d = size(X)` returns the sizes of each dimension of array `X` in a vector `d` with `ndims(X)` elements. If `X` is a scalar, which MATLAB software regards as a 1-by-1 array, `size(X)` returns the vector `[1 1]`.

`[m,n] = size(X)` returns the size of matrix `X` in separate variables `m` and `n`.

`m = size(X,dim)` returns the size of the dimension of `X` specified by scalar `dim`.

`[d1,d2,d3,...,dn] = size(X)`, for `n` > 1, returns the sizes of the dimensions of the array `X` in the variables d1,d2,d3,...,dn, provided the number of output arguments `n` equals `ndims(X)`. If `n` does not equal `ndims(X)`, the following exceptions hold:

`n < ndims(X)`	`di` equals the size of the `i`th dimension of `X` for `0<i<n`, but `dn` equals the product of the sizes of the remaining dimensions of `X`, that is, dimensions `n` through `ndims(X)`.
`n > ndims(X)`	`size` returns ones in the “extra” variables, that is, those corresponding to `ndims(X)+1` through `n`.

Note For a Java array, `size` returns the length of the Java array as the number of rows. The number of columns is always 1. For a Java array of arrays, the result describes only the top level array.

Examples

Example 1

The size of the second dimension of `rand(2,3,4)` is 3.

```
m = size(rand(2,3,4),2)

m =
     3
```

Here the size is output as a single vector.

```
d = size(rand(2,3,4))

d =
     2     3     4
```

Here the size of each dimension is assigned to a separate variable.

```
[m,n,p] = size(rand(2,3,4))
m =
     2

n =
     3

p =
     4
```

Example 2

If `X = ones(3,4,5)`, then

```
[d1,d2,d3] = size(X)

d1 =        d2 =        d3 =
     3           4           5
```

But when the number of output variables is less than `ndims(X)`:

```
[d1,d2] = size(X)

d1 =        d2 =
     3           20
```

The "extra" dimensions are collapsed into a single product.

If `n > ndims(X)`, the "extra" variables all represent singleton dimensions:

```
[d1,d2,d3,d4,d5,d6] = size(X)

d1 =        d2 =        d3 =
     3           4           5

d4 =        d5 =        d6 =
     1           1           1
```

See Also

`exist` | `length` | `numel` | `whos`

Purpose

Size of `containers.Map` object

Syntax

```
dim = size(mapObj,1)
dimVector = size(mapObj)
[dim1,dim2,...,dimN] = size(mapObj)
```

Description

`dim = size(mapObj,1)` returns a scalar numeric value that indicates the number of key-value pairs in `mapObj`. If you call `size` with a numeric second input argument other than `1`, the `size` method returns the scalar numeric value `1`.

`dimVector = size(mapObj)` returns a two-element vector `[k,1]`, where `k` is the number of key-value pairs in `mapObj`.

`[dim1,dim2,...,dimN] = size(mapObj)` returns `[k,1,...,1]`.

Input Arguments

mapObj

Object of class `containers.Map`.

Output Arguments

dim

Scalar numeric value that indicates the number of key-value pairs in `mapObj`.

dimVector

Two-element numeric vector `[k,1]`, where `k` is the number of key-value pairs in `mapObj`.

[dim1,dim2,...,dimN]

Numeric scalar values. Variable `dim1` equals `k`, where `k` is the number of key-value pairs in `mapObj`. All other outputs equal `1`.

Examples

Determine the Size of a Map

Construct a map and find the number of key-value pairs:

```
myKeys = {'a','b','c'};
myValues = [1,2,3];
```

```
mapObj = containers.Map(myKeys,myValues);
dim = size(mapObj,1)
```

This code returns a scalar numeric value:

```
dim =
     3
```

If you do not specify a second input argument,

```
dimVector = size(mapObj)
```

then the `size` method returns a vector:

```
dimVector =
     3     1
```

See Also

`containers.Map` | `isKey` | `keys` | `length` | `values`

Purpose

Array dimensions

Syntax

```
allDims = size(matObj,variable)
[dim1,...,dimN] = size(matObj,variable)
selectedDim = size(matObj,variable,dim)
```

Description

`allDims = size(matObj,variable)` returns the size of each dimension of the specified variable in the file corresponding to `matObj`. Output `allDims` is a `1`-by-`m` vector, where `m = ndims(variable)`.

`[dim1,...,dimN] = size(matObj,variable)` returns the sizes of each dimension in separate output variables `dim1,...,dimN`.

`selectedDim = size(matObj,variable,dim)` returns the size of the specified dimension.

Tips

- Do not call `size` with the syntax `size(matObj.variable)`. This syntax loads the entire contents of the variable into memory. For very large variables, this load operation results in `Out of Memory` errors.

Input Arguments

matObj

Object created by the `matfile` function.

variable

String enclosed in single quotation marks that specifies the name of a variable in the MAT-file corresponding to `matObj`.

dim

Nonzero positive scalar integer that specifies a dimension of the variable.

Output Arguments

allDims

`1`-by-`m` vector of sizes of the dimensions of the specified variable, where `m = ndims(variable)`.

dim1,...,dimN

Scalar numeric values, where dimK contains the size of the Kth dimension of variable:

- If N < ndims(variable), then dimN, equals the product of the sizes of dimensions N through ndims(variable).
- If N > ndims(variable), the size method returns ones in the output variables corresponding to dimensions ndims(variable)+1 through N.

selectedDim

Scalar numeric value that contains the size of the selected dimension for the specified variable.

Examples

Find the size of the matrix topo in topography.mat without loading any data:

```
matObj = matfile('topography.mat');
[nrows,ncols] = size(matObj,'topo');
```

Determine the dimensions of a variable, and process one part of the variable at a time. In this case, calculate and store the average of each column of variable stocks in the example file stocks.mat:

```
filename = 'stocks.mat';
matObj = matfile(filename);
[nrows, ncols] = size(matObj,'stocks');

avgs = zeros(1,ncols);
for idx = 1:ncols
    avgs(idx) = mean(matObj.stocks(:,idx));
end
```

Create a three-dimensional array, and call the size method with different numbers of output arguments:

```
matObj = matfile('temp.mat','Writable',true);
matObj.X = rand(2,3,4);

d = size(matObj,'X')
d2 = size(matObj,'X',2)
[m,n] = size(matObj,'X')
[m1,m2,m3,m4] = size(matObj,'X')
```

This code returns

```
d =
     2     3     4

d2 =
     3

m =
     2
n =
    12

m1 =
     2
m2 =
     3
m3 =
     4
m4 =
     1
```

See Also

matfile | whos

size (serial)

Purpose

Size of serial port object array

Syntax

```
d = size(obj)
[m,n] = size(obj)
[m1,m2,m3,...,mn] = size(obj)
m = size(obj,dim)
```

Description

`d = size(obj)` returns the two-element row vector `d` containing the number of rows and columns in the serial port object, `obj`.

`[m,n] = size(obj)` returns the number of rows, `m` and columns, `n` in separate output variables.

`[m1,m2,m3,...,mn] = size(obj)` returns the length of the first `n` dimensions of `obj`.

`m = size(obj,dim)` returns the length of the dimension specified by the scalar `dim`. For example, `size(obj,1)` returns the number of rows.

See Also

`length`

Purpose

(Will be removed) Size of triangulation matrix

Note `size(TriRep)` will be removed in a future release. Use `size(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

`size(TR)`

Description

`size(TR)` provides size information for a triangulation matrix. The matrix is of size `mtri`-by-`nv`, where `mtri` is the number of simplices and `nv` is the number of vertices per simplex (triangle/tetrahedron, etc).

Input Arguments

`TR`	Triangulation matrix

Definitions

A simplex is a triangle/tetrahedron or higher-dimensional equivalent.

See Also

`size` | `delaunayTriangulation` | `triangulation`

size (tscollection)

Purpose Size of `tscollection` object

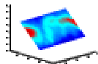
Syntax `size(tsc)`

Description `size(tsc)` returns `[n m]`, where `n` is the length of the time vector and `m` is the number of `tscollection` members.

See Also `length (tscollection)` | `isempty (tscollection)` | `tscollection`

Purpose

Volumetric slice plot



Syntax

```
slice(V,sx,sy,sz)
slice(X,Y,Z,V,sx,sy,sz)
slice(V,XI,YI,ZI)
slice(X,Y,Z,V,XI,YI,ZI)
slice(...,'method')
slice(axes_handle,...)
h = slice(...)
```

Description

`slice` displays orthogonal slice planes through volumetric data.

`slice(V,sx,sy,sz)` draws slices along the *x*, *y*, *z* directions in the volume `V` at the points in the vectors `sx`, `sy`, and `sz`. `V` is an *m*-by-*n*-by-*p* volume array containing data values at the default location `X = 1:n`, `Y = 1:m`, `Z = 1:p`. Each element in the vectors `sx`, `sy`, and `sz` defines a slice plane in the *x*-, *y*-, or *z*-axis direction.

`slice(X,Y,Z,V,sx,sy,sz)` draws slices of the volume `V`. `X`, `Y`, and `Z` are three-dimensional arrays specifying the coordinates for `V`. `X`, `Y`, and `Z` must be monotonic and orthogonally spaced (as if produced by the function `meshgrid`). The color at each point is determined by 3-D interpolation into the volume `V`.

`slice(V,XI,YI,ZI)` draws data in the volume `V` for the slices defined by `XI`, `YI`, and `ZI`. `XI`, `YI`, and `ZI` are matrices that define a surface, and the volume is evaluated at the surface points. `XI`, `YI`, and `ZI` must all be the same size.

`slice(X,Y,Z,V,XI,YI,ZI)` draws slices through the volume `V` along the surface defined by the arrays `XI`, `YI`, `ZI`.

`slice(...,'method')` specifies the interpolation method. `'method'` is `'linear'`, `'cubic'`, or `'nearest'`.

- `linear` specifies trilinear interpolation (the default).

- `cubic` specifies tricubic interpolation.
- `nearest` specifies nearest-neighbor interpolation.

`slice(axes_handle,...)` plots into the axes with the handle `axes_handle` instead of into the current axes object (`gca`). The axes `clim` property is set to span the finite values of `V`.

`h = slice(...)` returns a vector of handles to surface graphics objects.

Tips

The color drawn at each point is determined by interpolation into the volume `V`.

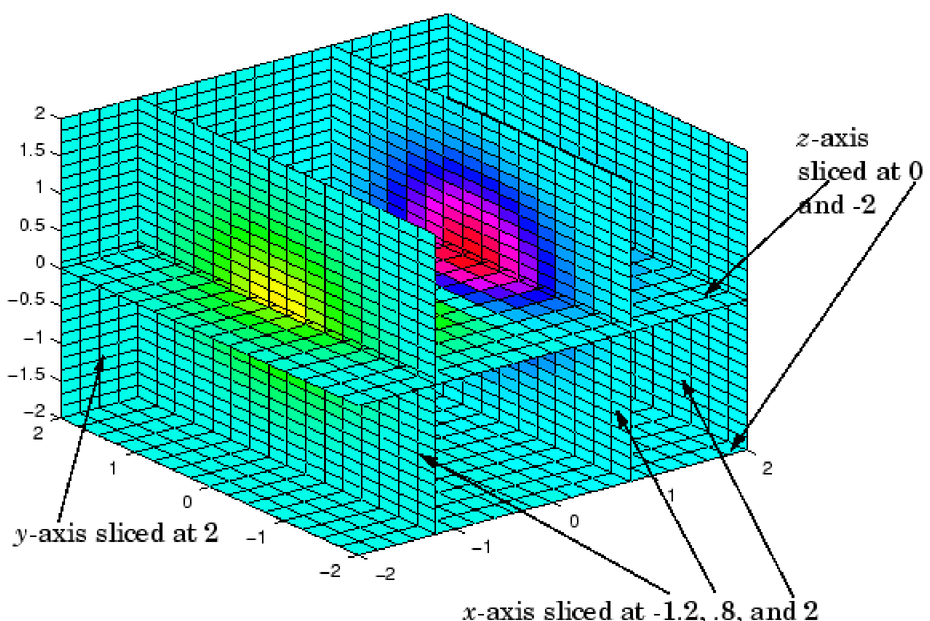
Examples

Visualize the function

$$v = xe^{\left(-x^2 - y^2 - z^2\right)}$$

over the range $-2 \le x \le 2$, $-2 \le y \le 2$, $-2 \le z \le 2$:

```
[x,y,z] = meshgrid(-2:.2:2,-2:.25:2,-2:.16:2);
v = x.*exp(-x.^2-y.^2-z.^2);
xslice = [-1.2,.8,2]; yslice = 2; zslice = [-2,0];
slice(x,y,z,v,xslice,yslice,zslice)
colormap hsv
```



Slicing At Arbitrary Angles

You can also create slices that are oriented in arbitrary planes. To do this,

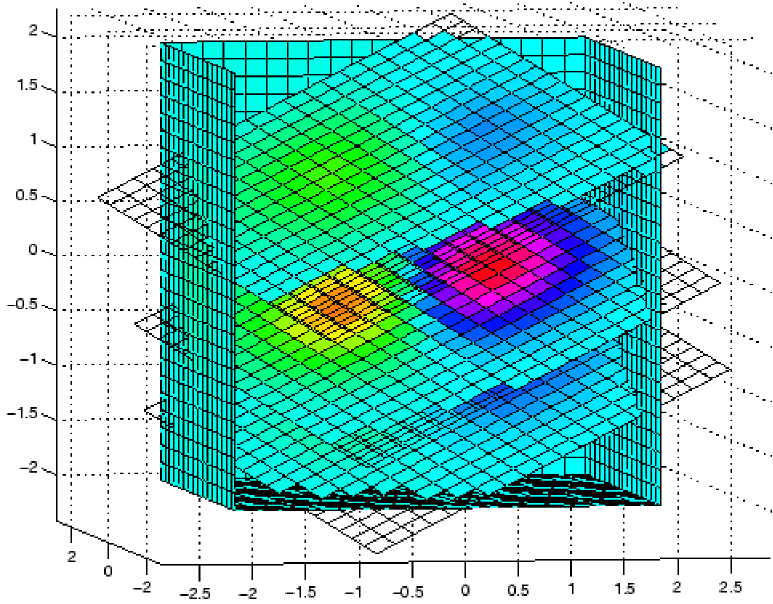
- Create a slice surface in the domain of the volume (`surf`, `linspace`).
- Orient this surface with respect to the axes (`rotate`).
- Get the `XData`, `YData`, and `ZData` of the surface (`get`).
- Use this data to draw the slice plane within the volume.

For example, these statements slice the volume in the first example with a rotated plane. Placing these commands within a `for` loop "passes" the plane through the volume along the *z*-axis.

```
for i = -2:.5:2
 hsp = surf(linspace(-2,2,20),linspace(-2,2,20),zeros(20)+i);
 rotate(hsp,[1,-1,1],30)
```

```
    xd = get(hsp,'XData');
    yd = get(hsp,'YData');
    zd = get(hsp,'ZData');
    delete(hsp)
    slice(x,y,z,v,[-2,2],2,-2) % Draw some volume boundaries
    hold on
    slice(x,y,z,v,xd,yd,zd)
    hold off
    axis tight
    view(-5,10)
    drawnow
end
```

The following picture illustrates three positions of the same slice surface as it passes through the volume.

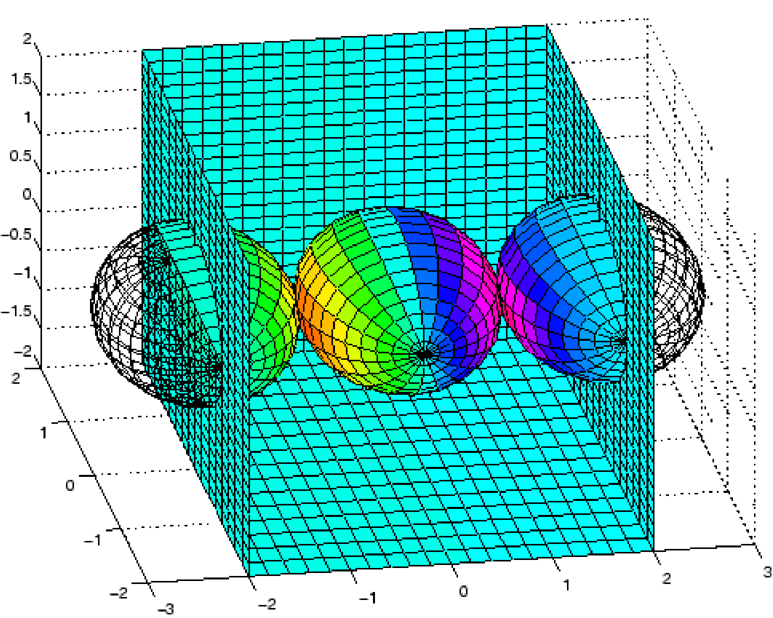


Slicing with a Nonplanar Surface

You can slice the volume with any surface. This example probes the volume created in the previous example by passing a spherical slice surface through the volume.

```
[xsp,ysp,zsp] = sphere;
slice(x,y,z,v,[-2,2],2,-2)  % Draw some volume boundaries

for i = -3:.2:3
 hsp = surface(xsp+i,ysp,zsp);
 rotate(hsp,[1 0 0],90)
 xd = get(hsp,'XData');
 yd = get(hsp,'YData');
 zd = get(hsp,'ZData');
 delete(hsp)
 hold on
 hslicer = slice(x,y,z,v,xd,yd,zd);
 axis tight
 xlim([-3,3])
 view(-10,35)
 drawnow
 delete(hslicer)
 hold off
end
```

The following picture illustrates three positions of the spherical slice surface as it passes through the volume.



See Also

`interp3` | `meshgrid`

How To

- Exploring Volumes with Slice Planes

Purpose

Smooth 3-D data

Syntax

Description

W = smooth3(V) smooths the input data V and returns the smoothed data in W.

W = smooth3(V,'*filter*') *filter* determines the convolution kernel and can be the strings

- 'gaussian'
- 'box' (default)

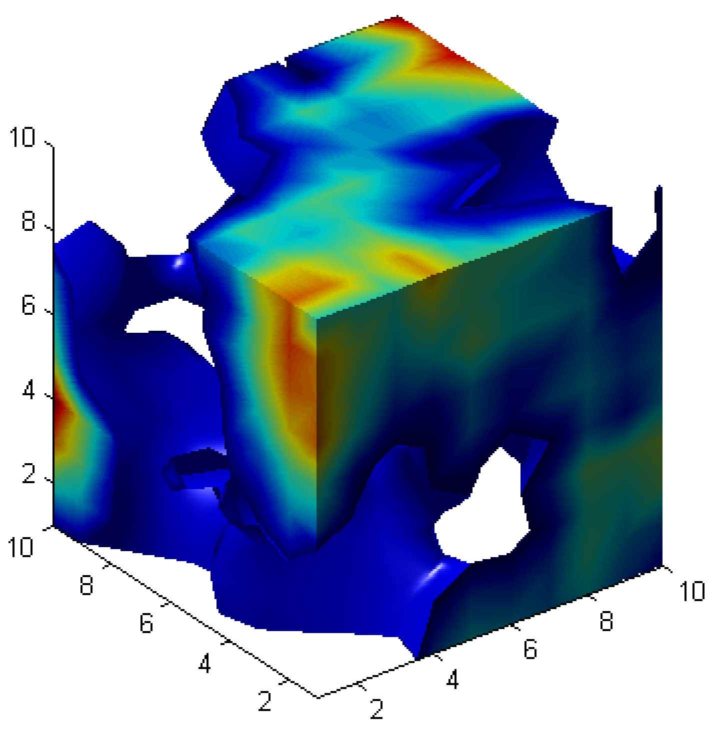
W = smooth3(V,'*filter*',size) sets the size of the convolution kernel (default is [3 3 3]). If size is scalar, then size is interpreted as [size, size, size].

W = smooth3(V,'*filter*',size,sd) sets an attribute of the convolution kernel. When *filter* is gaussian, sd is the standard deviation (default is .65).

Examples

This example smooths some random 3-D data and then creates an isosurface with end caps.

```
rng(9,'twister')
data = rand(10,10,10);
data = smooth3(data,'box',5);
patch(isocaps(data,.5),'FaceColor','interp','EdgeColor','none');
p1 = patch(isosurface(data,.5),'FaceColor','blue','EdgeColor','none');
isonormals(data,p1)
view(3);
axis vis3d tight
camlight left;
lighting phong
```



See Also

isocaps | isonormals | isosurface | patch

How To

- Displaying an Isosurface

Purpose

Force snapshot of image for inclusion in published document

Syntax

```
snapnow
```

Description

The `snapnow` command forces a snapshot of the image or plot that the code has most recently generated for presentation in a published document. The output appears in the published document at the end of the cell that contains the `snapnow` command. When used outside the context of publishing a file, `snapnow` has the same behavior as `drawnow`. That is, if you run a file that contains the `snapnow` command, the MATLAB software interprets it as though it were a `drawnow` command.

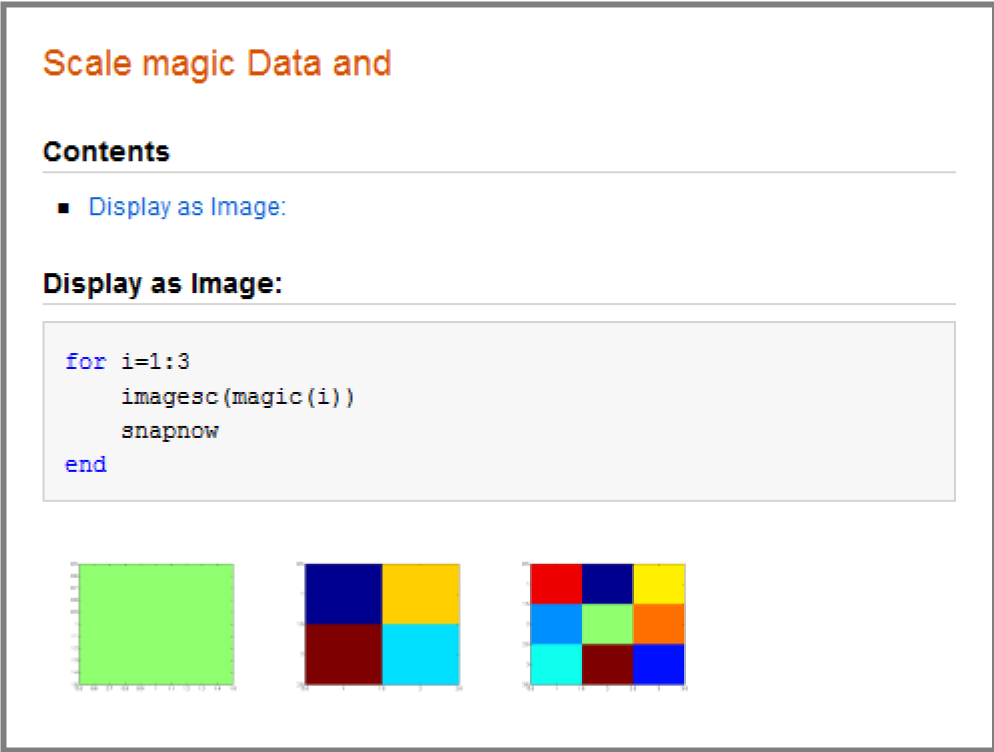
Examples

This example demonstrates the difference between publishing code that contains the `snapnow` command and running that code. The first image shows the results of publishing the code and the second image shows the results of running the code.

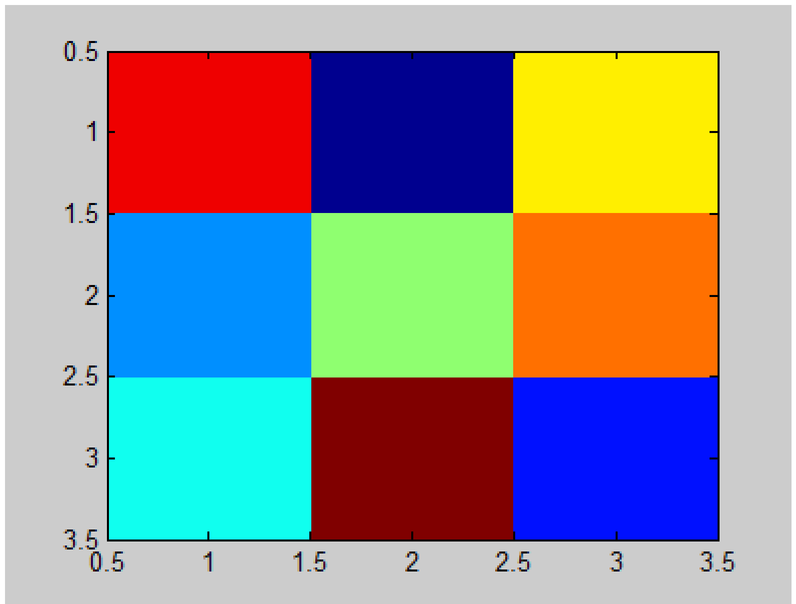
Suppose you have a file that contains the following code:

```
%% Scale magic Data and
%% Display as Image:

for i=1:3
    imagesc(magic(i))
    snapnow
end
```

When you publish the code to HTML, the published document contains a title, a table of contents, the commented text, the code, and each of the three images produced by the `for` loop. (In the published document shown, the size of the images have been reduced.)



When you run the code, a single Figure window opens and MATLAB updates the image within this window as it evaluates each iteration of the `for` loop. Each successive image replaces the one that preceded it, so that the Figure window appears as follows when the code evaluation completes.



See Also

drawnow

Concepts

- "Image Snapshot"

Purpose

Sort array elements in ascending or descending order

Syntax

```
B = sort(A)
B = sort(A,dim)
B = sort(...,mode)
[B,IX] = sort(A,...)
```

Description

`B = sort(A)` sorts the elements along different dimensions of an array, and arranges those elements in ascending order.

If A is a ...	sort(A) ...
Vector	Sorts the elements of `A`.
Matrix	Sorts each column of `A`.
Multidimensional array	Sorts `A` along the first non-singleton dimension, and returns an array of sorted vectors.
Cell array of strings	Sorts the strings in ascending ASCII dictionary order, and returns a vector cell array of strings. The sort is case-sensitive; uppercase letters appear in the output before lowercase. You cannot use the `dim` or `mode` options with a cell array.

Integer, floating-point, logical, and character arrays are permitted. Floating-point arrays can be complex. For elements of `A` with identical values, the order of these elements is preserved in the sorted list. When `A` is complex, the elements are sorted by magnitude, i.e., `abs(A)`, and where magnitudes are equal, further sorted by phase angle, i.e., `angle(A)`, on the interval $[-\pi, \pi]$. If `A` includes any `NaN` elements, `sort` places these at the high end.

`B = sort(A,dim)` sorts the elements along the dimension of `A` specified by a scalar `dim`.

`B = sort(...,mode)` sorts the elements in the specified direction, depending on the value of `mode`.

'ascend'	Ascending order (default)
'descend'	Descending order

[B,IX] = sort(A,...) also returns an array of indices IX, where size(IX) == size(A). If A is a vector, B = A(IX). If A is an m-by-n matrix, then each column of IX is a permutation vector of the corresponding column of A, such that

```
for j = 1:n
    B(:,j) = A(IX(:,j),j);
end
```

If A has repeated elements of equal value, the returned indices preserve the original ordering.

Sorting Complex Entries

If A has complex entries r and s, sort orders them according to the following rule: r appears before s in sort(A) if either of the following hold:

- abs(r) < abs(s)
- abs(r) = abs(s) and angle(r)<angle(s)

  where $-\pi < angle(r) \leq \pi$

For example,

```
v = [1 -1 i -i];
angle(v)

ans =

        0    3.1416    1.5708   -1.5708
sort(v)

ans =

       0 - 1.0000i   1.0000
```

```
      0 + 1.0000i  -1.0000
```

Note `sort` uses a different rule for ordering complex numbers than do the relational operators. See the Relational Operators reference page for more information.

Examples

Example 1

Sort horizontal vector `A`:

```
A = [78 23 10 100 45 5 6];

sort(A)
ans =
     5     6    10    23    45    78   100
```

Example 2

Sort matrix `A` in each dimension:

```
A = [ 3 7 5
      0 4 2 ];

sort(A,1)

ans =
     0     4     2
     3     7     5

sort(A,2)

ans =
     3     5     7
     0     2     4
```

Sort it again, this time returning an array of indices for the result:

```
[B, IX] = sort(A, 2)

B =
     3     5     7
     0     2     4

IX =
     1     3     2
     1     3     2
```

Example 3

Sort each column of matrix A in descending order:

```
A = [ 3  7  5
      6  8  3
      0  4  2 ];

sort(A,1,'descend')

ans =
     6   8   5
     3   7   3
     0   4   2
```

This is equivalent to

```
sort(A,'descend')

ans =
     6   8   5
     3   7   3
     0   4   2
```

See Also

issorted | max | mean | median | min | sortrows | unique

sortrows

Purpose

Sort rows in ascending order

Syntax

```
B = sortrows(A)
B = sortrows(A,column)
[B,index] = sortrows(A,...)
```

Description

`B = sortrows(A)` sorts the rows of `A` in ascending order. Argument `A` must be either a matrix or a column vector.

For strings, this is the familiar dictionary sort. When `A` is complex, the elements are sorted by magnitude, and, where magnitudes are equal, further sorted by phase angle on the interval $[-\pi, \pi]$.

`B = sortrows(A,column)` sorts the matrix based on the columns specified in the vector `column`. If an element of `column` is positive, the MATLAB software sorts the corresponding column of matrix `A` in ascending order; if an element of `column` is negative, MATLAB sorts the corresponding column in descending order. For example, `sortrows(A,[2 -3])` sorts the rows of `A` first in ascending order for the second column, and then by descending order for the third column.

`[B,index] = sortrows(A,...)` also returns an index vector `index`.

If `A` is a column vector, then `B = A(index)`. If `A` is an `m`-by-`n` matrix, then `B = A(index,:)`.

Examples

Start with an arbitrary matrix, `A`:

```
A=floor(gallery('uniformdata',[6 7],0)*100);
A(1:4,1)=95;  A(5:6,1)=76;  A(2:4,2)=7;  A(3,3)=73
A =
    95    45    92    41    13     1    84
    95     7    73    89    20    74    52
    95     7    73     5    19    44    20
    95     7    40    35    60    93    67
    76    61    93    81    27    46    83
    76    79    91     0    19    41     1
```

When called with only a single input argument, `sortrows` bases the sort on the first column of the matrix. For any rows that have equal elements in a particular column, (e.g., `A(1:4,1)` for this matrix), sorting is based on the column immediately to the right, (`A(1:4,2)` in this case):

```
B = sortrows(A)
B =
    76    61    93    81    27    46    83
    76    79    91     0    19    41     1
    95     7    40    35    60    93    67
    95     7    73     5    19    44    20
    95     7    73    89    20    74    52
    95    45    92    41    13     1    84
```

When called with two input arguments, `sortrows` bases the sort entirely on the column specified in the second argument. Rows that have equal elements in the specified column, (e.g., `A(2:4,:)`, if sorting matrix `A` by column 2) remain in their original order:

```
C = sortrows(A,2)
C =
    95     7    73    89    20    74    52
    95     7    73     5    19    44    20
    95     7    40    35    60    93    67
    95    45    92    41    13     1    84
    76    61    93    81    27    46    83
    76    79    91     0    19    41     1
```

This example specifies two columns to sort by: columns 1 and 7. This tells `sortrows` to sort by column 1 first, and then for any rows with equal values in column 1, to sort by column 7:

```
D = sortrows(A,[1 7])
D =
    76    79    91     0    19    41     1
    76    61    93    81    27    46    83
    95     7    73     5    19    44    20
    95     7    73    89    20    74    52
```

```
    95     7    40    35    60    93    67
    95    45    92    41    13     1    84
```

Sort the matrix using the values in column 4 this time and in reverse order:

```
E = sortrows(A, -4)
E =
    95     7    73    89    20    74    52
    76    61    93    81    27    46    83
    95    45    92    41    13     1    84
    95     7    40    35    60    93    67
    95     7    73     5    19    44    20
    76    79    91     0    19    41     1
```

See Also

`issorted` | `sort`

Purpose

Convert matrix of signal data to sound

Syntax

```
sound(y)
sound(y,Fs)
sound(y,Fs,nBits)
```

Description

`sound(y)` sends audio signal `y` to the speaker at the default sample rate of 8192 hertz.

`sound(y,Fs)` sends audio signal `y` to the speaker at sample rate `Fs`.

`sound(y,Fs,nBits)` uses `nBits` bits per sample for audio signal `y`.

Input Arguments

y - Audio data

column vector | `m`-by-2 matrix

Audio data, specified as an `m`-by-1 column vector for single-channel (mono) audio, or an `m`-by-2 matrix for stereo playback, where `m` is the number of audio samples. If `y` is an `m`-by-2 matrix, then the first column corresponds to the left channel, and the second column corresponds to the right channel. Stereo playback is available only if your system supports it.

Data Types
`double`

Fs - Sample rate

8192 (default) | positive number

Sample rate, in hertz, of audio data `y`, specified as a positive number between 80 and 1000000.

Data Types
`single | double`

nBits - Bit depth of sample values

8 (default) | 16 | 24

Bit depth of the sample values, specified as an integer. Valid values depend on the audio hardware installed. Most platforms support bit depths of 8 bits or 16 bits.

Examples

Play Sample Data at Default Sample Rate

Load the example file `gong.mat`, which contains sample data `y` and rate `Fs`, and listen to the audio.

```
load gong.mat;
sound(y);
```

Play Sample Data at Specific Sample Rate

Play an excerpt from Handel's "Hallelujah Chorus" at twice the recorded sample rate.

```
load handel.mat;
sound(y, 2*Fs);
```

Play Sample Data with Specific Bit Depth

```
load handel.mat;
nBits = 16;
sound(y,Fs,nBits);
```

MATLAB plays the audio with a bit depth of 16 bits per sample, if this is supported on your system.

Tips

- The `sound` function supports sound devices on all Windows and most UNIX platforms.
- Most sound cards support sample rates between 5 and 48 kilohertz. Specifying a sample rate outside this range might produce unexpected results.

See Also

`audioplayer` | `soundsc` | `audioread` | `audiowrite`

Concepts

- "Characteristics of Audio Files"
- "Play Audio"

soundsc

Purpose

Scale data and play as sound

Syntax

```
soundsc(y)
soundsc(y,Fs)
soundsc(y,Fs,nBits)

soundsc( ___ ,yRange)
```

Description

`soundsc(y)` scales the values of audio signal `y` to fit in the range from –1.0 to 1.0, and then sends the data to the speaker at the default sample rate of 8192 hertz. By first scaling the data, `soundsc` plays the audio as loudly as possible without clipping. The mean of the dynamic range of the data is set to zero.

`soundsc(y,Fs)` sends audio signal `y` to the speaker at sample rate `Fs`.

`soundsc(y,Fs,nBits)` uses `nBits` bits per sample for audio signal `y`.

`soundsc( ___ ,yRange)`, where `yRange` is a vector of the form `[low,high]`, linearly scales the values in `y` between `low` and `high` to the full sound range `[-1.0,1.0]`. Values outside `[low,high]` are not clipped. You can use `yRange` with any of the input arguments in the above syntaxes.

Input Arguments

y - Audio data

column vector | m-by-2 matrix

Audio data, specified as an `m`-by-1 column vector for single-channel (mono) audio, or an `m`-by-2 matrix for stereo playback, where `m` is the number of audio samples. If `y` is an `m`-by-2 matrix, then the first column corresponds to the left channel, and the second column corresponds to the right channel. Stereo playback is available only if your system supports it.

Data Types

`double`

Fs - Sample rate

8192 (default) | positive number

Sample rate, in hertz, of audio data `y`, specified as a positive number between 80 and 1000000.

Data Types
`single | double`

nBits - Bit depth of sample values

8 (default) | 16 | 24

Bit depth of the sample values, specified as an integer. Valid values depend on the audio hardware installed. Most platforms support bit depths of 8 bits or 16 bits.

yRange - Range of audio data to scale

`[min(y),max(y)]` (default) | two-element vector

Range of audio data to scale, specified as a two-element vector of the form `[low,high]`, where `low` and `high` are the lower and upper limits of the range.

Example: `[-0.8,0.8]`

Data Types
`double`

Examples

Play Sample Data at Default Sample Rate

Load the example file gong.mat, which contains sample data `y` and rate `Fs`, and listen to the audio.

```
load gong.mat;
soundsc(y);
```

Play Sample Data at Specific Sample Rate

Play an excerpt from Handel's "Hallelujah Chorus" at twice the recorded sample rate.

```
load handel.mat;
soundsc(y, 2*Fs);
```

Play Sample Data with Specific Bit Depth

```
load handel.mat;
nBits = 16;
soundsc(y,Fs,nBits);
```

MATLAB plays the scaled audio with a bit depth of 16 bits per sample.

Scale Selected Audio Data

```
load handel.mat;
yRange = [-0.7,0.7];
soundsc(y,yRange);
```

Tips

- The `sound` function supports sound devices on all Windows and most UNIX platforms.
- Most sound cards support sample rates between 5 and 48 kilohertz. Specifying a sample rate outside this range might produce unexpected results.

See Also

`audioplayer` | `sound` | `audioread` | `audiowrite`

Purpose

Allocate space for sparse matrix

Syntax

```
S = spalloc(m,n,nzmax)
```

Description

`S = spalloc(m,n,nzmax)` creates an all zero sparse matrix `S` of size `m`-by-`n` with room to hold `nzmax` nonzeros. The matrix can then be generated column by column without requiring repeated storage allocation as the number of nonzeros grows.

`spalloc(m,n,nzmax)` is shorthand for

```
sparse([],[],[],m,n,nzmax)
```

Examples

To generate efficiently a sparse matrix that has an average of at most three nonzero elements per column

```
S = spalloc(n,n,3*n);
for j = 1:n
    S(:,j) = [zeros(n-3,1)' round(rand(3,1))']';end
```

sparse

Purpose

Create sparse matrix

Syntax

```
S = sparse(A)
S = sparse(i,j,s,m,n,nzmax)
S = sparse(i,j,s,m,n)
S = sparse(i,j,s)
S = sparse(m,n)
```

Description

The `sparse` function generates matrices in the MATLAB sparse storage organization.

`S = sparse(A)` converts a full matrix to sparse form by squeezing out any zero elements. If `S` is already sparse, `sparse(S)` returns `S`.

`S = sparse(i,j,s,m,n,nzmax)` uses vectors `i`, `j`, and `s` to generate an `m`-by-`n` sparse matrix such that `S(i(k),j(k)) = s(k)`, with space allocated for `nzmax` nonzeros. Vectors `i`, `j`, and `s` are all the same length. Any elements of `s` that are zero are ignored, along with the corresponding values of `i` and `j`. Any elements of `s` that have duplicate values of `i` and `j` are added together.

Note If any value in `i` or `j` is larger than 2^31-1 for 32-bit platforms, or 2^48-1 on 64-bit platforms, then the sparse matrix cannot be constructed.

To simplify this six-argument call, you can pass scalars for the argument `s` and one of the arguments `i` or `j`—in which case they are expanded so that `i`, `j`, and `s` all have the same length.

`S = sparse(i,j,s,m,n)` uses `nzmax = length(s)`.

`S = sparse(i,j,s)` uses `m = max(i)` and `n = max(j)`. The maxima are computed before any zeros in `s` are removed, so one of the rows of `[i j s]` might be `[m n 0]`.

`S = sparse(m,n)` abbreviates `sparse([],[],[],m,n,0)`. This generates the ultimate sparse matrix, an `m`-by-`n` all zero matrix.

Tips

All of the MATLAB built-in arithmetic, logical, and indexing operations can be applied to sparse matrices, or to mixtures of sparse and full matrices. Operations on sparse matrices return sparse matrices and operations on full matrices return full matrices.

In most cases, operations on mixtures of sparse and full matrices return full matrices. The exceptions include situations where the result of a mixed operation is structurally sparse, for example, `A.*S` is at least as sparse as `S`.

Examples

`S = sparse(1:n,1:n,1)` generates a sparse representation of the `n`-by-`n` identity matrix. The same `S` results from `S = sparse(eye(n,n))`, but this would also temporarily generate a full `n`-by-`n` matrix with most of its elements equal to zero.

`B = sparse(10000,10000,pi)` is probably not very useful, but is legal and works; it sets up a `10000`-by-`10000` matrix with only one nonzero element. Don't try `full(B)`; it requires 800 megabytes of storage.

This dissects and then reassembles a sparse matrix:

```
[i,j,s] = find(S);
[m,n] = size(S);
S = sparse(i,j,s,m,n);
```

So does this, if the last row and column have nonzero entries:

```
[i,j,s] = find(S);
S = sparse(i,j,s);
```

See Also

`diag` | `find` | `full` | `issparse` | `nnz` | `nonzeros` | `nzmax` | `spones` | `sprandn` | `sprandsym` | `spy`

spaugment

Purpose

Form least squares augmented system

Syntax

```
S = spaugment(A,c)
S = spaugment(A)
```

Description

`S = spaugment(A,c)` creates the sparse, square, symmetric indefinite matrix `S = [c*I A; A' 0]`. The matrix `S` is related to the least squares problem

```
min norm(b - A*x)
```

by

```
r = b - A*x
S * [r/c; x] = [b; 0]
```

The optimum value of the residual scaling factor `c`, involves `min(svd(A))` and `norm(r)`, which are usually too expensive to compute.

`S = spaugment(A)` without a specified value of `c`, uses `max(max(abs(A)))/1000`.

Note In previous versions of MATLAB product, the augmented matrix was used by sparse linear equation solvers, \ and /, for nonsquare problems. Now, MATLAB software performs a least squares solve using the `qr` factorization of `A` instead.

See Also

`spparms`

Purpose

Import matrix from sparse matrix external format

Syntax

```
S = spconvert(D)
```

Description

`spconvert` is used to create sparse matrices from a simple sparse format easily produced by non-MATLAB sparse programs. `spconvert` is the second step in the process:

1. Load an ASCII data file containing `[i,j,v]` or `[i,j,re,im]` as rows into a MATLAB variable.
2. Convert that variable into a MATLAB sparse matrix.

`S = spconvert(D)` converts a matrix `D` with rows containing `[i,j,s]` or `[i,j,r,s]` to the corresponding sparse matrix. `D` must have an `nnz` or `nnz+1` row and three or four columns. Three elements per row generate a real matrix and four elements per row generate a complex matrix. A row of the form `[m n 0]` or `[m n 0 0]` anywhere in `D` can be used to specify `size(S)`. If `D` is already sparse, no conversion is done, so `spconvert` can be used after `D` is loaded from either a MAT-file or an ASCII file.

Examples

Suppose the ASCII file `uphill.dat` contains

```
1      1      1.000000000000000
1      2      0.500000000000000
2      2      0.333333333333333
1      3      0.333333333333333
2      3      0.250000000000000
3      3      0.200000000000000
1      4      0.250000000000000
2      4      0.200000000000000
3      4      0.166666666666667
4      4      0.142857142857143
4      4      0.000000000000000
```

Then the statements

```
load uphill.dat
H = spconvert(uphill)

H =
   (1,1)       1.0000
   (1,2)       0.5000
   (2,2)       0.3333
   (1,3)       0.3333
   (2,3)       0.2500
   (3,3)       0.2000
   (1,4)       0.2500
   (2,4)       0.2000
   (3,4)       0.1667
   (4,4)       0.1429
```

recreate `sparse(triu(hilb(4)))`, possibly with roundoff errors. In this case, the last line of the input file is not necessary because the earlier lines already specify that the matrix is at least 4-by-4.

Purpose

Extract and create sparse band and diagonal matrices

Syntax

```
B = spdiags(A)
[B,d] = spdiags(A)
B = spdiags(A,d)
A = spdiags(B,d,A)
A = spdiags(B,d,m,n)
```

Description

The `spdiags` function generalizes the function `diag`. Four different operations, distinguished by the number of input arguments, are possible.

`B = spdiags(A)` extracts all nonzero diagonals from the `m`-by-`n` matrix `A`. `B` is a `min(m,n)`-by-`p` matrix whose columns are the `p` nonzero diagonals of `A`.

`[B,d] = spdiags(A)` returns a vector `d` of length `p`, whose integer components specify the diagonals in `A`.

`B = spdiags(A,d)` extracts the diagonals specified by `d`.

`A = spdiags(B,d,A)` replaces the diagonals specified by `d` with the columns of `B`. The output is sparse.

`A = spdiags(B,d,m,n)` creates an `m`-by-`n` sparse matrix by taking the columns of `B` and placing them along the diagonals specified by `d`.

Note In this syntax, if a column of `B` is longer than the diagonal it is replacing, and `m >= n`, `spdiags` takes elements of super-diagonals from the lower part of the column of `B`, and elements of sub-diagonals from the upper part of the column of `B`. However, if `m < n` , then super-diagonals are from the upper part of the column of `B`, and sub-diagonals from the lower part. (See “Example 5A” on page 1-4805 and “Example 5B” on page 1-4807, below).

Arguments

The `spdiags` function deals with three matrices, in various combinations, as both input and output.

A	An m-by-n matrix, usually (but not necessarily) sparse, with its nonzero or specified elements located on p diagonals.
B	A min(m,n)-by-p matrix, usually (but not necessarily) full, whose columns are the diagonals of A.
d	A vector of length p whose integer components specify the diagonals in A.

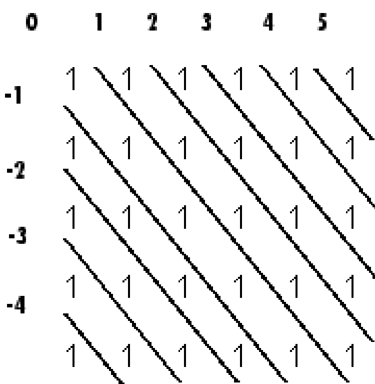
Roughly, A, B, and d are related by

```
for k = 1:p
    B(:,k) = diag(A,d(k))
end
```

Some elements of B, corresponding to positions outside of A, are not defined by these loops. They are not referenced when B is input and are set to zero when B is output.

How the Diagonals of A are Listed in the Vector d

An m-by-n matrix A has m+n-1diagonals. These are specified in the vector d using indices from -m+1 to n-1. For example, if A is 5-by-6, it has 10 diagonals, which are specified in the vector d using the indices -4, -3 , ... 4, 5. The following diagram illustrates this for a vector of all ones.



Examples

Example 1

For the following matrix,

```
A=[0 5 0 10 0 0;...
0 0 6 0 11 0;...
3 0 0 7 0 12;...
1 4 0 0 8 0;...
0 2 5 0 0 9]

A =

     0     5     0    10     0     0
     0     0     6     0    11     0
     3     0     0     7     0    12
     1     4     0     0     8     0
     0     2     5     0     0     9
```

the command

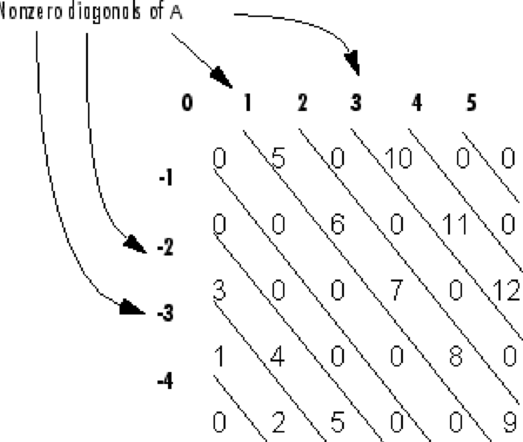
```
[B, d]  =spdiags(A)
```

returns

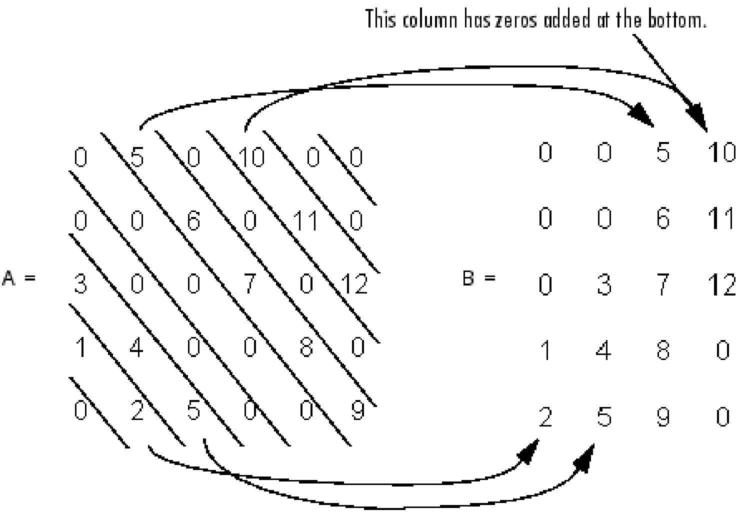
```
B =

     0     0     5    10
     0     0     6    11
     0     3     7    12
     1     4     8     0
     2     5     9     0


d =


    -3
    -2
     1
```

```
    3
```

The columns of the first output B contain the nonzero diagonals of A. The second output d lists the indices of the nonzero diagonals of A, as shown in the following diagram. See "How the Diagonals of A are Listed in the Vector d" on page 1-4800.



Note that the longest nonzero diagonal in A is contained in column 3 of B. The other nonzero diagonals of A have extra zeros added to their corresponding columns in B, to give all columns of B the same length. For the nonzero diagonals below the main diagonal of A, extra zeros are added at the tops of columns. For the nonzero diagonals above the main diagonal of A, extra zeros are added at the bottoms of columns. This is illustrated by the following diagram.



Example 2

This example generates a sparse tridiagonal representation of the classic second difference operator on `n` points.

```
e = ones(n,1);
A = spdiags([e -2*e e], -1:1, n, n)
```

Turn it into Wilkinson's test matrix (see `gallery`):

```
A = spdiags(abs(-(n-1)/2:(n-1)/2)',0,A)
```

Finally, recover the three diagonals:

```
B = spdiags(A)
```

Example 3

The second example is not square.

```
A = [11    0   13    0
      0   22    0   24
```

```
     0     0    33     0
    41     0     0    44
     0    52     0     0
     0     0    63     0
     0     0     0    74]
```

Here `m =7`, `n = 4`, and `p = 3`.

The statement `[B,d] = spdiags(A)` produces `d = [-3 0 2]'` and

```
B = [41    11     0
     52    22     0
     63    33    13
     74    44    24]
```

Conversely, with the above `B` and `d`, the expression `spdiags(B,d,7,4)` reproduces the original `A`.

Example 4

This example shows how `spdiags` creates the diagonals when the columns of `B` are longer than the diagonals they are replacing.

```
B = repmat((1:6)',[1 7])

B =

     1   1   1   1   1   1   1
     2   2   2   2   2   2   2
     3   3   3   3   3   3   3
     4   4   4   4   4   4   4
     5   5   5   5   5   5   5
     6   6   6   6   6   6   6

d = [-4 -2 -1 0 3 4 5];
A = spdiags(B,d,6,6);
full(A)

ans =
```

```
1  0  0  4  5  6
1  2  0  0  5  6
1  2  3  0  0  6
0  2  3  4  0  0
1  0  3  4  5  0
0  2  0  4  5  6
```

Example 5A

This example illustrates the use of the syntax `A = spdiags(B,d,m,n)`, under three conditions:

- `m` is equal to `n`
- `m` is greater than `n`
- `m` is less than `n`

The command used in this example is

```
A = full(spdiags(B, [-2 0 2], m, n))
```

where `B` is the 5-by-3 matrix shown below. The resulting matrix `A` has dimensions `m`-by-`n`, and has nonzero diagonals at `[-2 0 2]` (a sub-diagonal at `-2`, the main diagonal, and a super-diagonal at `2`).

```
B =
    1     6    11
    2     7    12
    3     8    13
    4     9    14
    5    10    15
```

The first and third columns of matrix `B` are used to create the sub- and super-diagonals of `A` respectively. In all three cases though, these two outer columns of `B` are longer than the resulting diagonals of `A`. Because of this, only a part of the columns is used in `A`.

When `m == n` or `m > n`, `spdiags` takes elements of the super-diagonal in `A` from the lower part of the corresponding column of `B`, and elements

of the sub-diagonal in A from the upper part of the corresponding column of B.

When m < n, spdiags does the opposite, taking elements of the super-diagonal in A from the upper part of the corresponding column of B, and elements of the sub-diagonal in A from the lower part of the corresponding column of B.

Part 1 — m is equal to n.

```
A = full(spdiags(B, [-2 0 2], 5, 5))
 Matrix B                             Matrix A

1     6    11                      6     0    13     0     0
2     7    12                      0     7     0    14     0
3     8    13   == spdiags =>      1     0     8     0    15
4     9    14                      0     2     0     9     0
5    10    15                      0     0     3     0    10
```

A(3,1), A(4,2), and A(5,3) are taken from the upper part of B(:,1).

A(1,3), A(2,4), and A(3,5) are taken from the lower part of B(:,3).

Part 2 — m is greater than n.

```
A = full(spdiags(B, [-2 0 2], 5, 4))
 Matrix B                             Matrix A

1     6    11                      6     0    13     0
2     7    12                      0     7     0    14
3     8    13   == spdiags =>      1     0     8     0
4     9    14                      0     2     0     9
5    10    15                      0     0     3     0
```

Same as in Part A.

Part 3 — m is less than n.

```
A = full(spdiags(B, [-2 0 2], 4, 5))
```

```
 Matrix B                          Matrix A

1    6   11                      6    0   11    0    0
2    7   12                      0    7    0   12    0
3    8   13  == spdiags =>       3    0    8    0   13
4    9   14                      0    4    0    9    0
5   10   15
```

`A(3,1)` and `A(4,2)` are taken from the lower part of `B(:,1)`.

`A(1,3)`, `A(2,4)`, and `A(3,5)` are taken from the upper part of `B(:,3)`.

Example 5B

Extract the diagonals from the first part of this example back into a column format using the command

```
B = spdiags(A)
```

You can see that in each case the original columns are restored (minus those elements that had overflowed the super- and sub-diagonals of matrix A).

Part 1.

```
      Matrix A                               Matrix B

6    0   13    0    0                   1    6    0
0    7    0   14    0                   2    7    0
1    0    8    0   15  == spdiags =>    3    8   13
0    2    0    9    0                   0    9   14
0    0    3    0   10                   0   10   15
```

Part 2.

```
    Matrix A                           Matrix B

6    0   13    0                    1    6    0
0    7    0   14                    2    7    0
1    0    8    0  == spdiags =>     3    8   13
```

```
0     2     0     9                    0    9   14
0     0     3     0
```

Part 3.

```
      Matrix A                              Matrix B

6     0    11     0     0                   0    6   11
0     7     0    12     0                   0    7   12
3     0     8     0    13  == spdiags =>    3    8   13
0     4     0     9     0                   4    9    0
```

See Also

diag | speye

Purpose

Calculate specular reflectance

Syntax

```
R = specular(Nx,Ny,Nz,S,V)
```

Description

`R = specular(Nx,Ny,Nz,S,V)` returns the reflectance of a surface with normal vector components `[Nx,Ny,Nz]`. `S` and `V` specify the direction to the light source and to the viewer, respectively. You can specify these directions as three vectors`[x,y,z]` or two vectors `[Theta Phi` (in spherical coordinates).

The specular highlight is strongest when the normal vector is in the direction of `(S+V)/2` where `S` is the source direction, and `V` is the view direction.

The surface spread exponent can be specified by including a sixth argument as in `specular(Nx,Ny,Nz,S,V,spread)`.

speye

Purpose

Sparse identity matrix

Syntax

```
S = speye(m,n)
S = speye([m n])
S = speye(n)
S = speye
```

Description

`S = speye(m,n)` and `S = speye([m n])` form an `m`-by-`n` sparse matrix with `1`s on the main diagonal.

`S = speye(n)` abbreviates `speye(n,n)`.

`S = speye` returns the sparse form of the 1-by-1 identity matrix.

Examples

`I = speye(1000)` forms the sparse representation of the 1000-by-1000 identity matrix, which requires only about 16 kilobytes of storage. This is the same final result as `I = sparse(eye(1000,1000))`, but the latter requires eight megabytes for temporary storage for the full representation.

See Also

`spalloc` | `spones` | `spdiags` | `sprand` | `sprandn`

Purpose

Apply function to nonzero sparse matrix elements

Syntax

```
f = spfun(fun,S)
```

Description

The `spfun` function selectively applies a function to only the *nonzero* elements of a sparse matrix `S`, preserving the sparsity pattern of the original matrix (except for underflow or if `fun` returns zero for some nonzero elements of `S`).

`f = spfun(fun,S)` evaluates `fun(S)` on the elements of `S` that are nonzero. `fun` is a function handle.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

Tips

Functions that operate element-by-element, like those in the `elfun` directory, are the most appropriate functions to use with `spfun`.

Examples

Given the 4-by-4 sparse diagonal matrix

```
S = spdiags([1:4]',0,4,4)

S =
   (1,1)        1
   (2,2)        2
   (3,3)        3
   (4,4)        4
```

Because `fun` returns nonzero values for all nonzero element of `S`, `f = spfun(@exp,S)` has the same sparsity pattern as `S.`

```
f =
   (1,1)       2.7183
   (2,2)       7.3891
   (3,3)      20.0855
   (4,4)      54.5982
```

whereas `exp(S)` has `1`s where `S` has `0`s.

```
full(exp(S))

ans =
    2.7183    1.0000    1.0000    1.0000
    1.0000    7.3891    1.0000    1.0000
    1.0000    1.0000   20.0855    1.0000
    1.0000    1.0000    1.0000   54.5982
```

See Also

function_handle

Purpose

Transform spherical coordinates to Cartesian

Syntax

```
[x,y,z] = sph2cart(azimuth,elevation,r)
```

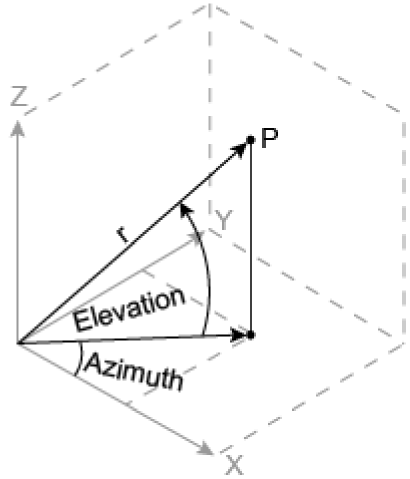
Description

`[x,y,z] = sph2cart(azimuth,elevation,r)` transforms the corresponding elements of spherical coordinate arrays to Cartesian, or *xyz*, coordinates. `azimuth`, `elevation`, and `r` must all be the same size (or any of them can be scalar). `azimuth` and `elevation` are angular displacements in radians from the positive *x*-axis and from the *x*-*y* plane, respectively.

Algorithms

The mapping from spherical coordinates to three-dimensional Cartesian coordinates is

```
x = r .* cos(elevation) .* cos(azimuth)
y = r .* cos(elevation) .* sin(azimuth)
z = r .* sin(elevation)
```



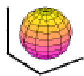
See Also

`cart2pol` | `cart2sph` | `pol2cart`

sphere

Purpose

Generate sphere



Syntax

```
sphere
sphere(n)
[X,Y,Z] = sphere(n)
```

Description

The `sphere` function generates the *x*-, *y*-, and *z*-coordinates of a unit sphere for use with `surf` and `mesh`.

`sphere` generates a sphere consisting of 20-by-20 faces.

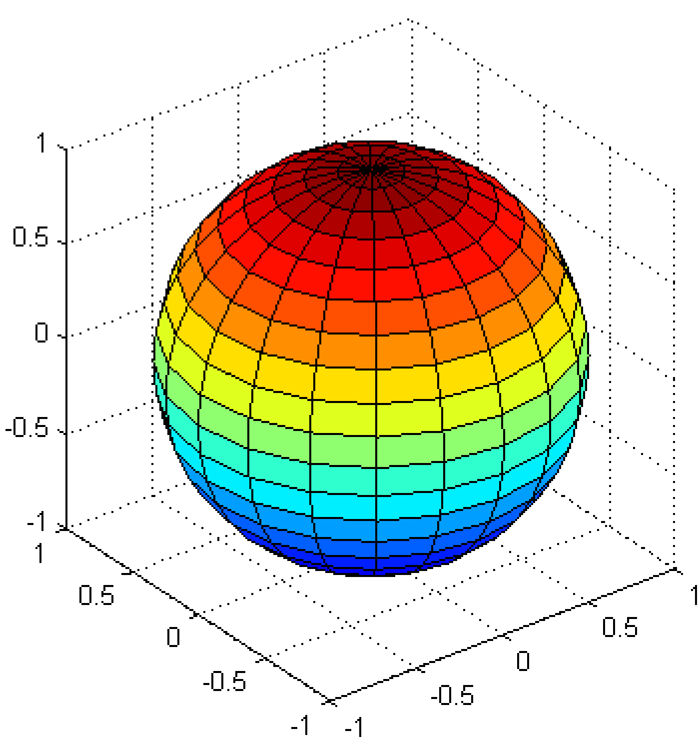
`sphere(n)` draws a `surf` plot of an `n`-by-`n` sphere in the current figure.

`[X,Y,Z] = sphere(n)` returns the coordinates of a sphere in three matrices that are `(n+1)`-by-`(n+1)` in size. You draw the sphere with `surf(X,Y,Z)` or `mesh(X,Y,Z)`.

Examples

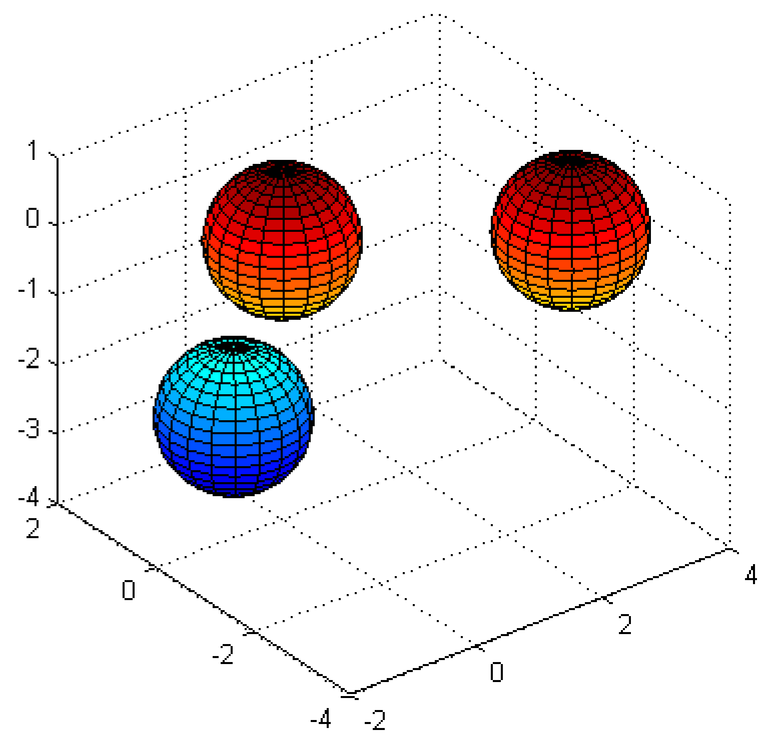
Generate and plot a sphere.

```
figure
sphere
axis equal
```



Plot multiple spheres, translating centers away from the origin:

```
figure
[x,y,z] = sphere;
surf(x,y,z)  % sphere centered at origin
hold on
surf(x+3,y-2,z)  % sphere centered at (3,-2,0)
surf(x,y+1,z-3)  % sphere centered at (0,1,-3)
daspect([1 1 1])
```



See Also

`cylinder` | `axis`

spinmap

Purpose

Spin colormap

Syntax

```
spinmap
spinmap(t)
spinmap(t,inc)
spinmap('inf')
```

Description

The `spinmap` function shifts the colormap RGB values by some incremental value. For example, if the increment equals 1, color 1 becomes color 2, color 2 becomes color 3, etc.

`spinmap` cyclically rotates the colormap for approximately five seconds using an incremental value of 2.

`spinmap(t)` rotates the colormap for approximately 10*t seconds. The amount of time specified by `t` depends on your hardware configuration (e.g., if you are running MATLAB software over a network).

`spinmap(t,inc)` rotates the colormap for approximately 10*t seconds and specifies an increment `inc` by which the colormap shifts. When `inc` is 1, the rotation appears smoother than the default (i.e., 2). Increments greater than 2 are less smooth than the default. A negative increment (e.g., –2) rotates the colormap in a negative direction.

`spinmap('inf')` rotates the colormap for an infinite amount of time. To break the loop, press **Ctrl+C**.

See Also

`colormap` | `colormapeditor`

spline

Purpose

Cubic spline data interpolation

Syntax

```
yy = spline(x,Y,xx)
pp = spline(x,Y)
```

Description

`yy = spline(x,Y,xx)` uses a cubic spline interpolation to find `yy`, the values of the underlying function `Y` at the values of the interpolant `xx`. For the interpolation, the independent variable is assumed to be the final dimension of `Y` with the breakpoints defined by `x`.

The sizes of `xx` and `yy` are related as follows:

- If `Y` is a scalar or vector, `yy` has the same size as `xx`.
- If `Y` is an array that is not a vector,
  - If `xx` is a scalar or vector, `size(yy)` equals `[d1, d2, ..., dk, length(xx)]`.
  - If `xx` is an array of size `[m1,m2,...,mj]`, `size(yy)` equals `[d1,d2,...,dk,m1,m2,...,mj]`.

`pp = spline(x,Y)` returns the piecewise polynomial form of the cubic spline interpolant for later use with `ppval` and the spline utility `unmkpp`. `x` must be a vector. `Y` can be a scalar, a vector, or an array of any dimension, subject to the following conditions:

- If x and Y are vectors of the same size, the not-a-knot end conditions are used.
- If x or Y is a scalar, it is expanded to have the same length as the other and the not-a-knot end conditions are used. (See Exceptions (1) below).
- If Y is a vector that contains two more values than x has entries, the first and last value in Y are used as the endslopes for the cubic spline. (See Exceptions (2) below.)

Exceptions

1. If `Y` is a vector that contains two more values than `x` has entries, the first and last value in `Y` are used as the endslopes for the cubic spline. If `Y` is a vector, this means
   - `f(x) = Y(2:end-1)`
   - `df(min(x)) = Y(1)`
   - `df(max(x)) = Y(end)`
2. If `Y` is a matrix or an `N`-dimensional array with `size(Y,N)` equal to `length(x)+2`, the following hold:
   - `f(x(j))` matches the value `Y(:,...,:,j+1)` for `j=1:length(x)`
   - `Df(min(x))` matches `Y(:,:,...:,1)`
   - `Df(max(x))` matches `Y(:,:,...:,end)`

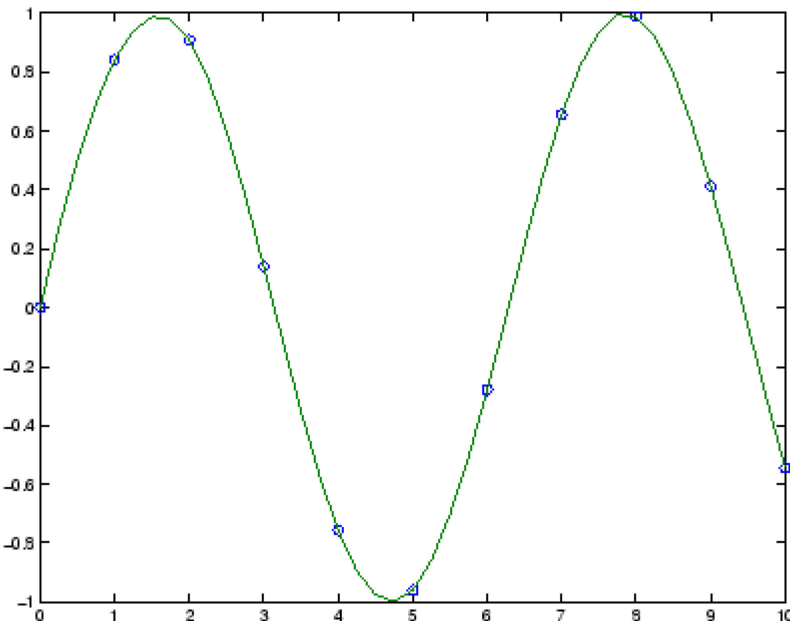
Note You can also perform spline interpolation using the `interp1` function with the command `interp1(x,y,xx,'spline')`. Note that while `spline` performs interpolation on rows of an input matrix, `interp1` performs interpolation on columns of an input matrix.

Examples

Example 1

This generates a sine curve, then samples the spline over a finer mesh.

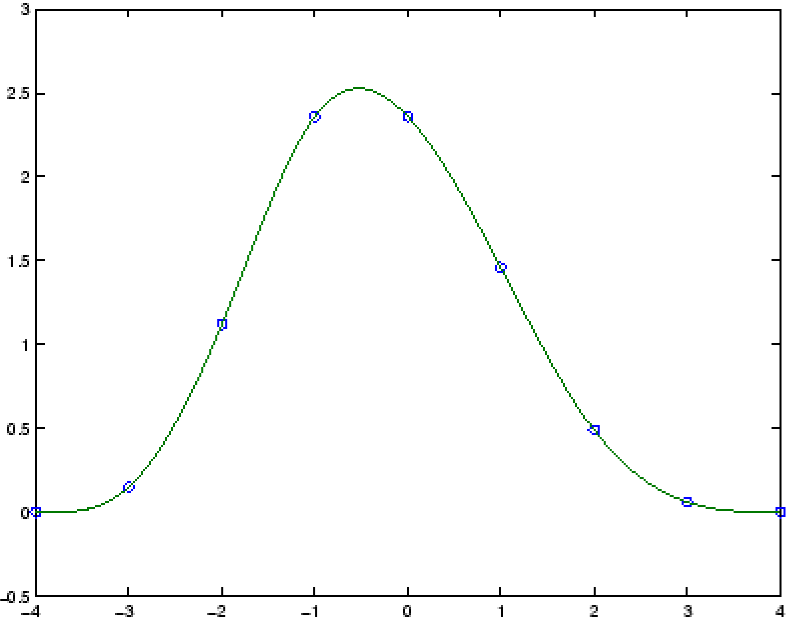
```
x = 0:10;
y = sin(x);
xx = 0:.25:10;
yy = spline(x,y,xx);
plot(x,y,'o',xx,yy)
```



Example 2

This illustrates the use of clamped or complete spline interpolation where end slopes are prescribed. Zero slopes at the ends of an interpolant to the values of a certain distribution are enforced.

```
x = -4:4;
y = [0 .15 1.12 2.36 2.36 1.46 .49 .06 0];
cs = spline(x,[0 y 0]);
xx = linspace(-4,4,101);
plot(x,y,'o',xx,ppval(cs,xx),'-');
```



Example 3

The two vectors

```
t = 1900:10:1990;
p = [ 75.995  91.972  105.711  123.203  131.669 ...
     150.697 179.323  203.212  226.505  249.633 ];
```

represent the census years from 1900 to 1990 and the corresponding United States population in millions of people. The expression

```
spline(t,p,2000)
```

uses the cubic spline to extrapolate and predict the population in the year 2000. The result is

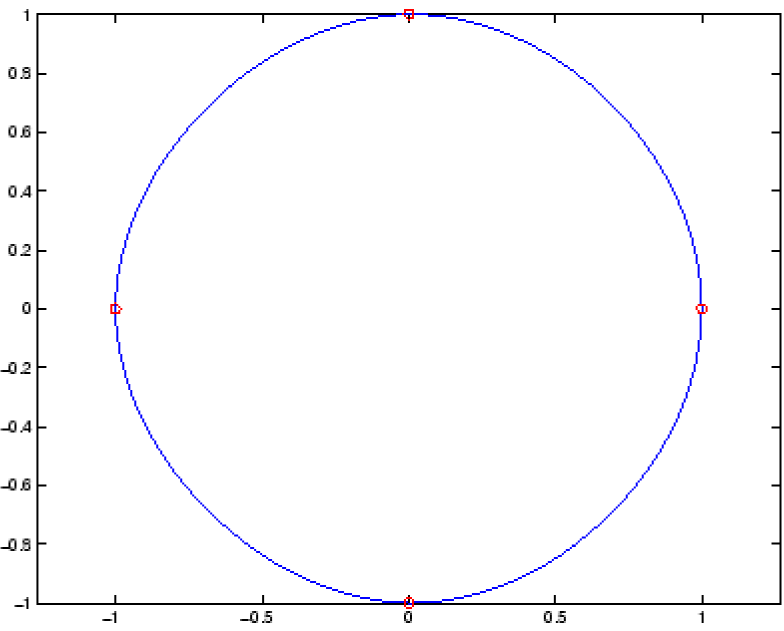
```
ans =
    270.6060
```

Example 4

The statements

```
x = pi*[0:.5:2];
y = [0  1  0 -1  0  1  0;
     1  0  1  0 -1  0  1];
pp = spline(x,y);
yy = ppval(pp, linspace(0,2*pi,101));
plot(yy(1,:),yy(2,:),'-b',y(1,2:5),y(2,2:5),'or'), axis equal
```

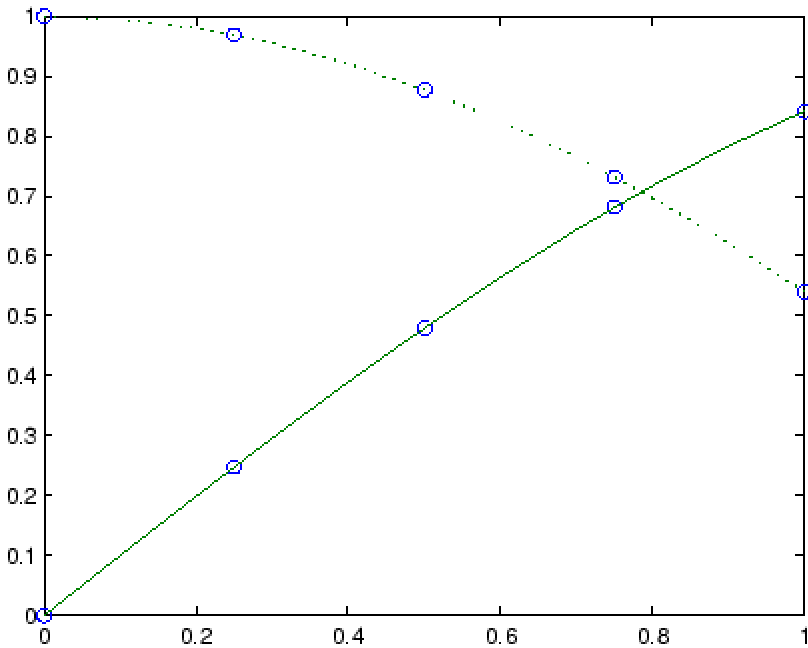
generate the plot of a circle, with the five data points `y(:,2),...,y(:,6)` marked with o's. Note that this `y` contains two more values (i.e., two more columns) than does `x`, hence `y(:,1)` and `y(:,end)` are used as endslopes.



Example 5

The following code generates sine and cosine curves, then samples the splines over a finer mesh.

```
x = 0:.25:1;
Y = [sin(x); cos(x)];
xx = 0:.1:1;
YY = spline(x,Y,xx);
plot(x,Y(1,:),'o',xx,YY(1,:),'-'); hold on;
plot(x,Y(2,:),'o',xx,YY(2,:),':'); hold off;
```



Algorithms

A tridiagonal linear system (with, possibly, several right sides) is being solved for the information needed to describe the coefficients of the various cubic polynomials which make up the interpolating spline. `spline` uses the functions `ppval`, `mkpp`, and `unmkpp`. These routines

form a small suite of functions for working with piecewise polynomials. For access to more advanced features, see the `interp1` reference page, the command-line help for these functions, and the Curve Fitting Toolbox spline functions.

References

[1] de Boor, C., *A Practical Guide to Splines*, Springer-Verlag, 1978.

See Also

`interp1` | `ppval` | `mkpp` | `pchip` | `unmkpp`

Purpose

Replace nonzero sparse matrix elements with ones

Syntax

`R = spones(S)`

Description

`R = spones(S)` generates a matrix `R` with the same sparsity structure as `S`, but with `1`'s in the nonzero positions.

Examples

`c = sum(spones(S))` is the number of nonzeros in each column.

`r = sum(spones(S'))'` is the number of nonzeros in each row.

`sum(c)` and `sum(r)` are equal, and are equal to `nnz(S)`.

See Also

`nnz` | `spalloc` | `spfun`

spparms

Purpose

Set parameters for sparse matrix routines

Syntax

```
spparms('key',value)
spparms
values = spparms
[keys,values] = spparms
spparms(values)
value = spparms('key')
spparms('default')
spparms('tight')
```

Description

`spparms('key',value)` sets one or more of the *tunable* parameters used in the sparse routines. In ordinary use, you should never need to deal with this function.

The meanings of the `key` parameters are

`'spumoni'`	Sparse Monitor flag:
`0`	Produces no diagnostic output, the default
`1`	Produces information about choice of algorithm based on matrix structure, and about storage allocation
`2`	Also produces very detailed information about the sparse matrix algorithms
`'thr_rel'`, `'thr_abs'`	Minimum degree threshold is `thr_rel*mindegree + thr_abs`.
`'exact_d'`	Nonzero to use exact degrees in minimum degree. Zero to use approximate degrees.
`'supernd'`	If positive, minimum degree amalgamates the supernodes every `supernd` stages.
`'rreduce'`	If positive, minimum degree does row reduction every `rreduce` stages.

`'wh_frac'`	Rows with `density > wh_frac` are ignored in `colmmd`.
`'autommd'`	Nonzero to use minimum degree (MMD) orderings with QR-based `\` and `/`.
`'autoamd'`	Nonzero to use `colamd` ordering with the LU-based \ and /, and to use `amd` with Cholesky-based \ and /.
`'piv_tol'`	Pivot tolerance used by the LU-based \ and /.
`'bandden'`	Band density used by `\` and `/` for banded matrices. Band density is defined as (# nonzeros in the band)/(# nonzeros in a full band). If `bandden = 1.0`, never use band solver. If `bandden = 0.0`, always use band solver. Default is `0.5`.
`'umfpack'`	Nonzero to use UMFPACK instead of the v4 LU-based solver in `\` and `/`.
`'sym_tol'`	Symmetric pivot tolerance. See `lu` for more information about the role of the symmetric pivot tolerance.

`spparms`, by itself, prints a description of the current settings.

`values = spparms` returns a vector whose components give the current settings.

`[keys,values] = spparms` returns that vector, and also returns a character matrix whose rows are the keywords for the parameters.

`spparms(values)`, with no output argument, sets all the parameters to the values specified by the argument vector.

`value = spparms('key')` returns the current setting of one parameter.

`spparms('default')` sets all the parameters to their default settings.

`spparms('tight')` sets the minimum degree ordering parameters to their *tight* settings, which can lead to orderings with less fill-in, but which make the ordering functions themselves use more execution time.

The key parameters for `default` and `tight` settings are

	Keyword	Default	Tight
values(1)	'spumoni'	0.0	
values(2)	'thr_rel'	1.1	1.0
values(3)	'thr_abs'	1.0	0.0
values(4)	'exact_d'	0.0	1.0
values(5)	'supernd'	3.0	1.0
values(6)	'rreduce'	3.0	1.0
values(7)	'wh_frac'	0.5	0.5
values(8)	'autommd'	1.0	
values(9)	'autoamd'	1.0	
values(10)	'piv_tol'	0.1	
values(11)	'bandden'	0.5	
values(12)	'umfpack'	1.0	
values(13)	'sym_tol'	0.001	

See Also

chol | lu | qr | colamd | symamd | Arithmetic Operator \

Purpose

Sparse uniformly distributed random matrix

Syntax

```
R = sprand(S)
R = sprand(m,n,density)
R = sprand(m,n,density,rc)
```

Description

`R = sprand(S)` has the same sparsity structure as `S`, but uniformly distributed random entries.

`R = sprand(m,n,density)` is a random, `m`-by-`n`, sparse matrix with approximately `density*m*n` uniformly distributed nonzero entries (`0 <= density <= 1`).

`R = sprand(m,n,density,rc)` also has reciprocal condition number approximately equal to `rc`. `R` is constructed from a sum of matrices of rank one.

If `rc` is a vector of length `lr`, where `lr <= min(m,n)`, then `R` has `rc` as its first `lr` singular values, all others are zero. In this case, `R` is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.

Tips

- `sprand` uses the same random number generator as `rand`, `randi`, and `randn`. You control this generator with `rng`.

See Also

`sprandn` | `sprandsym`

sprandn

Purpose

Sparse normally distributed random matrix

Syntax

```
R = sprandn(S)
R = sprandn(m,n,density)
R = sprandn(m,n,density,rc)
```

Description

`R = sprandn(S)` has the same sparsity structure as `S`, but normally distributed random entries with mean `0` and variance `1`.

`R = sprandn(m,n,density)` is a random, `m`-by-`n`, sparse matrix with approximately `density*m*n` normally distributed nonzero entries (`(0 <= density <= 1`).

`R = sprandn(m,n,density,rc)` also has reciprocal condition number approximately equal to `rc`. `R` is constructed from a sum of matrices of rank one.

If `rc` is a vector of length `lr`, where `lr <= min(m,n)`, then `R` has `rc` as its first `lr` singular values, all others are zero. In this case, `R` is generated by random plane rotations applied to a diagonal matrix with the given singular values. It has a great deal of topological and algebraic structure.

Tips

- `sprandn` uses the same random number generator as `rand`, `randi`, and `randn`. You control this generator with `rng`.

See Also

`sprand` | `sprandsym`

Purpose

Sparse symmetric random matrix

Syntax

```
R = sprandsym(S)
R = sprandsym(n,density)
R = sprandsym(n,density,rc)
R = sprandsym(n,density,rc,kind)
R = sprandsym(S,[],rc,3)
```

Description

`R = sprandsym(S)` returns a symmetric random matrix whose lower triangle and diagonal have the same structure as `S`. Its elements are normally distributed, with mean `0` and variance `1`.

`R = sprandsym(n,density)` returns a symmetric random, `n`-by-`n`, sparse matrix with approximately `density*n*n` nonzeros; each entry is the sum of one or more normally distributed random samples, and (`0 <= density <= 1`).

`R = sprandsym(n,density,rc)` returns a matrix with a reciprocal condition number equal to `rc`. The distribution of entries is nonuniform; it is roughly symmetric about 0; all are in [−1,1].

If `rc` is a vector of length `n`, then `R` has eigenvalues `rc`. Thus, if `rc` is a positive (nonnegative) vector then `R` is a positive (nonnegative) definite matrix. In either case, `R` is generated by random Jacobi rotations applied to a diagonal matrix with the given eigenvalues or condition number. It has a great deal of topological and algebraic structure.

`R = sprandsym(n,density,rc,kind)` is positive definite.

- If kind = 1, `R` is generated by random Jacobi rotation of a positive definite diagonal matrix. `R` has the desired condition number exactly.
- If kind = 2, `R` is a shifted sum of outer products. `R` has the desired condition number only approximately, but has less structure.

`R = sprandsym(S,[],rc,3)` has the same structure as the matrix `S` and approximate condition number `1/rc`.

Tips

`sprandsym` uses the same random number generator as `rand`, `randi`, and `randn`. You control this generator with `rng`.

See Also

`sprand` | `sprandn`

Purpose

Structural rank

Syntax

```
r = sprank(A)
```

Description

`r = sprank(A)` is the structural rank of the sparse matrix `A`. For all values of `A`,

```
sprank(A) >= rank(full(A))
```

In exact arithmetic, `sprank(A) == rank(full(sprandn(A)))` with a probability of one.

Examples

```
A = [1    0    2    0
     2    0    4    0 ];

A = sparse(A);

sprank(A)

ans =
     2

rank(full(A))

ans =
     1
```

See Also

`dmperm`

Purpose

Format data into string

Syntax

```
str = sprintf(formatSpec,A1,...,An)
[str,errmsg] = sprintf(formatSpec,A1,...,An)
```

Description

`str = sprintf(formatSpec,A1,...,An)` formats the data in arrays `A1,...,An` according to `formatSpec` in column order, and returns the results to string `str`.

`[str,errmsg] = sprintf(formatSpec,A1,...,An)` returns an error message string when the operation is unsuccessful. Otherwise, `errmsg` is empty.

Input Arguments

formatSpec - Format of the output fields

string

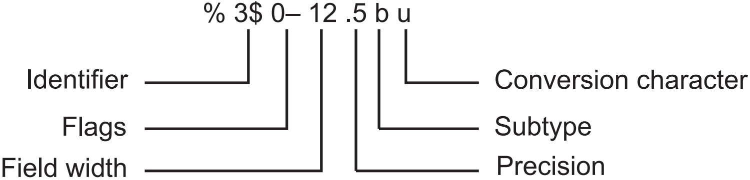
Format of the output fields, specified as a string.

The string can include a percent sign followed by a conversion character. The following table lists the available conversion characters and subtypes.

Value Type	Conversion	Details
Integer, signed	`%d` or `%i`	Base 10
Integer, unsigned	`%u`	Base 10
	`%o`	Base 8 (octal)
	`%x`	Base 16 (hexadecimal), lowercase letters `a`–`f`
	`%X`	Same as `%x`, uppercase letters `A`–`F`

Value Type	Conversion	Details
Floating-point number	%f	Fixed-point notation
	%e	Exponential notation, such as 3.141593e+00
	%E	Same as %e, but uppercase, such as 3.141593E+00
	%g	The more compact of %e or %f, with no trailing zeros
	%G	The more compact of %E or %f, with no trailing zeros
	%bx or %bX %bo %bu	Double-precision hexadecimal, octal, or decimal value Example: %bx prints pi as 400921fb54442d18
	%tx or %tX %to %tu	Single-precision hexadecimal, octal, or decimal value Example: %tx prints pi as 40490fdb
Characters	%c	Single character
	%s	String of characters

The string can include optional operators, which appear in the following order (includes spaces for clarity):



Optional operators include:

- Identifier

  Order for processing inputs. Use the syntax *n*$, where *n* represents the position of the value in the input list.

  For example, '%3$s %2$s %1$s %2$s' prints inputs 'A', 'B', 'C' as follows: C B A B.

- Flags

'-'	Left-justify. Example: %-5.2f
'+'	Print sign character (+) for positive values. Example: %+5.2f
' '	Pad to field width with spaces before the value. Example: % 5.2f
'0'	Pad to field width with zeros. Example: %05.2f
'#'	Modify selected numeric conversions: • For %o, %x, or %X, print 0, 0x, or 0X prefix. - For %f, %e, or %E, print decimal point even when precision is 0. - For %g or %G, do not remove trailing zeros or decimal point. Example: %#5.0f

- Field width

  Minimum number of characters to print. Can be a number, or an asterisk (*) to refer to an argument in the input list. For example, the input list ('%12d', intmax) is equivalent to ('%*d', 12, intmax).

- Precision

For `%f`, `%e`, or `%E`:	Number of digits to the right of the decimal point. Example: `'%6.4f'` prints `pi` as `'3.1416'`
For `%g` or `%G`	Number of significant digits. Example: `'%6.4g'` prints `pi` as `' 3.142'`

Can be a number, or an asterisk (`*`) to refer to an argument in the input list. For example, the input list `('%6.4f', pi)` is equivalent to `('%*.*f', 6, 4, pi)`.

The string can also include combinations of the following:

- Literal text to print. To print a single quotation mark, include `''` in `formatSpec`.
- Control characters, including:

`%%`	Percent character
`\\`	Backslash
`\a`	Alarm
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\xN`	Character whose ASCII code is the hexadecimal number, *N*
`\N`	Character whose ASCII code is the octal number, *N*

The following limitations apply to conversions:

- Numeric conversions print only the real component of complex numbers.
- If you specify a conversion that does not fit the data, such as a string conversion for a numeric value, MATLAB overrides the specified conversion, and uses `%e`.
- If you apply a string conversion (`%s`) to integer values, MATLAB converts values that correspond to valid character codes to characters. For example, `'%s'` converts `[65 66 67]` to `ABC`.

A1,...,An - Numeric or character arrays

scalar | vector | matrix | multidimensional array

Numeric or character arrays, specified as a scalar, vector, matrix, or multidimensional array.

Data Types
`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`
Complex Number Support: No

Output Arguments

str - Formatted text

string

Formatted text, returned as a string.

errmsg - Error message

string

Error message, returned as a string, when the operation is unsuccessful. Otherwise, `errmsg` is empty.

Tips

- The `sprintf` function is similar to `fprintf`, but `fprintf` prints to a file or to the Command Window.
- Format specifiers for the reading functions `sscanf` and `fscanf` differ from the formats for the writing functions `sprintf` and `fprintf`. The reading functions do not support a precision field. The width field specifies a minimum for writing but a maximum for reading.

Examples

Floating-Point Formats

Format a floating-point number using %e, %f, and %g specifiers.

```
A = 1/eps;
str_e = sprintf('%0.5e',A)
str_f = sprintf('%0.5f',A)
str_g = sprintf('%0.5g',A)
```

```
str_e =
4.50360e+15

str_f =
4503599627370496.00000

str_g =
4.5036e+15
```

Literal Text and Array Inputs

Combine literal text with array values to create a string.

```
formatSpec = 'The array is %dx%d.';
A1 = 2;
A2 = 3;
str = sprintf(formatSpec,A1,A2)
```

```
str =
The array is 2x3.
```

Integer Format with Floating-Point Inputs

Explicitly convert double-precision values to integers.

```
str = sprintf('%d',round(pi))
```

```
str =
3
```

Field Width

Specify the minimum width of the printed value.

```
str = sprintf('%025d',[123456])

str =
0000000000000000000123456
```

The 0 flag in the %025d format specifier requests leading zeros in the output.

Position Identifier (n$)

Reorder the input values using the n$ position identifier.

```
A1 = 'X';
A2 = 'Y';
A3 = 'Z';
formatSpec = ' %3$s %2$s %1$s';
str = sprintf(formatSpec,A1,A2,A3)

str =
 Z Y X
```

Cell Array Inputs

Create a string from values in a cell array.

```
C = { 1,   2,   3 ;
     'AA','BB','CC'};

str = sprintf(' %d %s',C{:})

str =
 1 AA 2 BB 3 CC
```

The syntax C{:} creates a comma-separated list of arrays that contain the contents of each cell from C in column order. For example, C{1}==1 and C{2}=='AA'.

References

[1] Kernighan, B. W., and D. M. Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, Inc., 1988.

[2] ANSI specification X3.159-1989: "Programming Language C," ANSI, 1430 Broadway, New York, NY 10018.

See Also

`char` | `fprintf` | `int2str` | `num2str` | `sscanf` | `fscanf`

Concepts

- "Formatting Strings"

spy

Purpose

Visualize sparsity pattern

Syntax

```
spy(S)
spy(S,markersize)
spy(S,'LineSpec')
spy(S,'LineSpec',markersize)
```

Description

plots the

`spy(S)` sparsity pattern of any matrix `S`.

`spy(S,markersize)`, where `markersize` is an integer, plots the sparsity pattern using markers of the specified point size.

`spy(S,'LineSpec')`, where `LineSpec` is a string, uses the specified plot marker type and color.

`spy(S,'LineSpec',markersize)` uses the specified type, color, and size for the plot markers.

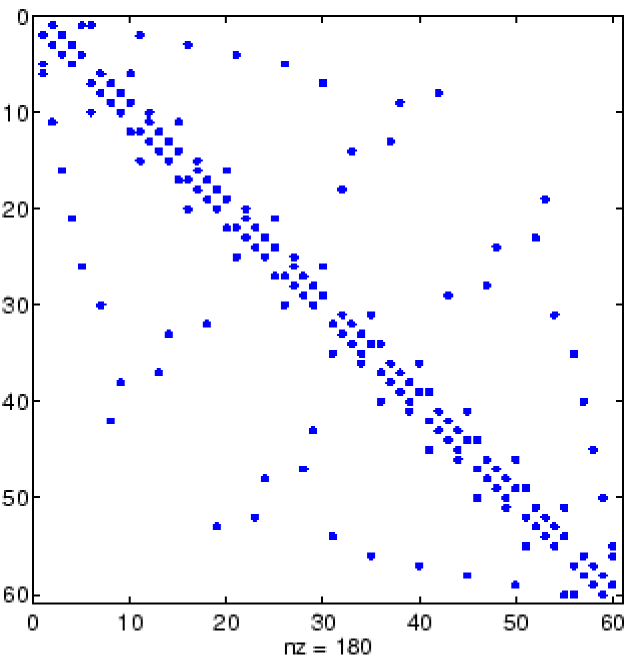
`S` is usually a sparse matrix, but full matrices are acceptable, in which case the locations of the nonzero elements are plotted.

Note `spy` replaces `format +`, which takes much more space to display essentially the same information.

Examples

This example plots the 60-by-60 sparse adjacency matrix of the connectivity graph of the Buckminster Fuller geodesic dome. This matrix also represents the soccer ball and the carbon-60 molecule.

```
B = bucky;
spy(B)
```



See Also

find | gplot | LineSpec | symamd | symrcm

sqrt

Purpose

Square root

Syntax

```
B = sqrt(X)
```

Description

`B = sqrt(X)` returns the square root of each element of the array `X`. For the elements of `X` that are negative or complex, `sqrt(X)` produces complex results.

Tips

See `sqrtm` for the matrix square root.

Examples

```
sqrt((-2:2)')
ans =
      0 + 1.4142i
      0 + 1.0000i
      0
 1.0000
 1.4142
```

See Also

`nthroot` | `sqrtm` | `realsqrt`

Purpose

Matrix square root

Syntax

```
X = sqrtm(A)
[X, resnorm] = sqrtm(A)
[X, alpha, condest] = sqrtm(A)
```

Description

`X = sqrtm(A)` is the principal square root of the matrix `A`, i.e. `X*X = A`.

`X` is the unique square root for which every eigenvalue has nonnegative real part. If `A` has any eigenvalues with negative real parts then a complex result is produced. If `A` is singular then `A` may not have a square root. A warning is printed if exact singularity is detected.

`[X, resnorm] = sqrtm(A)` does not print any warning, and returns the residual, `norm(A-X^2,'fro')/norm(A,'fro')`.

`[X, alpha, condest] = sqrtm(A)` returns a stability factor `alpha` and an estimate `condest` of the matrix square root condition number of `X`. The residual `norm(A-X^2,'fro')/norm(A,'fro')` is bounded approximately by `n*alpha*eps` and the Frobenius norm relative error in `X` is bounded approximately by `n*alpha*condest*eps`, where `n = max(size(A))`.

Tips

If `A` is real, symmetric and positive definite, or complex, Hermitian and positive definite, then so is the computed matrix square root.

Some matrices, like `A = [0 1; 0 0]`, do not have any square roots, real or complex, and `sqrtm` cannot be expected to produce one.

Examples

Example 1

A matrix representation of the fourth difference operator is

```
A =
     5    -4     1     0     0
    -4     6    -4     1     0
     1    -4     6    -4     1
     0     1    -4     6    -4
     0     0     1    -4     5
```

This matrix is symmetric and positive definite. Its unique positive definite square root, `Y = sqrtm(A)`, is a representation of the second difference operator.

```
 Y =
      2    -1    -0    -0    -0
     -1     2    -1     0    -0
      0    -1     2    -1     0
     -0     0    -1     2    -1
     -0    -0    -0    -1     2
```

Example 2

The matrix

```
A =
     7    10
    15    22
```

has four square roots. Two of them are

```
Y1 =
    1.5667    1.7408
    2.6112    4.1779
```

and

```
Y2 =
     1     2
     3     4
```

The other two are `-Y1` and `-Y2`. All four can be obtained from the eigenvalues and vectors of `A`.

```
[V,D] = eig(A);
D =
    0.1386         0
         0   28.8614
```

The four square roots of the diagonal matrix D result from the four choices of sign in

```
S =
    −0.3723          0
          0    −5.3723
```

All four Ys are of the form

```
Y = V*S/V
```

The `sqrtm` function chooses the two plus signs and produces Y1, even though Y2 is more natural because its entries are integers.

See Also

expm | funm | logm

squeeze

Purpose

Remove singleton dimensions

Syntax

```
B = squeeze(A)
```

Description

`B = squeeze(A)` returns an array `B` with the same elements as `A`, but with all singleton dimensions removed. A singleton dimension is any dimension for which `size(A,dim) = 1`. Two-dimensional arrays are unaffected by `squeeze`; if `A` is a row or column vector or a scalar (1-by-1) value, then `B = A`.

Examples

Consider the 2-by-1-by-3 array `Y = rand(2,1,3)`. This array has a singleton column dimension — that is, there's only one column per page.

```
Y =

Y(:,:,1) =       Y(:,:,2) =
       0.5194           0.0346
       0.8310           0.0535

Y(:,:,3) =
       0.5297
       0.6711
```

The command `Z = squeeze(Y)` yields a 2-by-3 matrix:

```
Z =
    0.5194    0.0346    0.5297
    0.8310    0.0535    0.6711
```

Consider the 1-by-1-by-5 array `mat=repmat(1,[1,1,5])`. This array has only one scalar value per page.

```
mat =

mat(:,:,1) =  mat(:,:,2) =

     1         1
```

```
mat(:,:,3) =   mat(:,:,4) =

     1         1

mat(:,:,5) =

     1
```

The command `squeeze(mat)` yields a 5-by-1 matrix:

```
squeeze(mat)

ans =

     1
     1
     1
     1
     1

size(squeeze(mat))

ans =

    5     1
```

See Also

`reshape` | `shiftdim`

Purpose

Convert state-space filter parameters to transfer function form

Syntax

```
[b,a] = ss2tf(A,B,C,D,iu)
```

Description

`ss2tf` converts a state-space representation of a given system to an equivalent transfer function representation.

`[b,a] = ss2tf(A,B,C,D,iu)` returns the transfer function

$$H(s) = \frac{B(s)}{A(s)} = C(sI - A)^{-1}B + D$$

of the system

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

from the `iu`-th input. Vector `a` contains the coefficients of the denominator in descending powers of s. The numerator coefficients are returned in array `b` with as many rows as there are outputs y. `ss2tf` also works with systems in discrete time, in which case it returns the z-transform representation.

The `ss2tf` function is part of the standard MATLAB language.

Purpose

Read formatted data from string

Syntax

```
A = sscanf(str, format)
A = sscanf(str, format, sizeA)
[A, count] = sscanf(...)
[A, count, errmsg] = sscanf(...)
[A, count, errmsg, nextindex] = sscanf(...)
```

Description

`A = sscanf(str, format)` reads data from string `str`, converts it according to the `format`, and returns the results in array `A`. The `sscanf` function reapplies the `format` until either reaching the end of `str` or failing to match the `format`. If `sscanf` cannot match the `format` to the data, it reads only the portion that matches into `A` and stops processing. If `str` is a character array with more than one row, `sscanf` reads the characters in column order.

`A = sscanf(str, format, sizeA)` reads `sizeA` elements into `A`, where `sizeA` can be an integer or can have the form `[m,n]`.

`[A, count] = sscanf(...)` returns the number of elements that `sscanf` successfully reads.

`[A, count, errmsg] = sscanf(...)` returns an error message string when the operation is unsuccessful. Otherwise, `errmsg` is an empty string.

`[A, count, errmsg, nextindex] = sscanf(...)` returns one more than the number of characters scanned in `str`.

Tips

- Format specifiers for the reading functions `sscanf` and `fscanf` differ from the formats for the writing functions `sprintf` and `fprintf`. The reading functions do not support a precision field. The width field specifies a minimum for writing but a maximum for reading.

Input Arguments

format

String enclosed in single quotation marks that describes each type of element (field). Includes one or more of the following specifiers.

Field Type	Specifier	Details
Integer, signed	`%d`	Base 10
	`%i`	Base determined from the values. Defaults to base 10. If initial digits are `0x` or `0X`, it is base 16. If initial digit is `0`, it is base 8.
	`%ld` or `%li`	64-bit values, base 10, 8, or 16
Integer, unsigned	`%u`	Base 10
	`%o`	Base 8 (octal)
	`%x`	Base 16 (hexadecimal)
	`%lu`, `%lo`, `%lx`	64-bit values, base 10, 8, or 16
Floating-point number	`%f`	Floating-point fields can contain any of the following (not case sensitive): `Inf`, `-Inf`, `NaN`, or `-NaN`.
	`%e`	
	`%g`	
Character string	`%s`	Read series of characters, until find white space.
	`%c`	Read any single character, including white space. (To read multiple characters, specify field length.)
	`%[...]`	Read only characters in the brackets, until the first nonmatching character or white space.

Optionally:

- To skip fields, insert an asterisk (`*`) after the percent sign (`%`). For example, to skip integers, specify `%*d`.

- To specify the maximum width of a field, insert a number. For example, `%10c` reads exactly 10 characters at a time, including white space.
- To skip a specific set of characters, insert the literal characters in the *format*. For example, to read only the floating-point number from `'pi=3.14159'`, specify a *format* of `'pi=%f'`.

sizeA

Dimensions of the output array *A*. Specify in one of the following forms:

`inf`	Read to the end of the input string. (default)
n	Read at most *n* elements.
`[`*m*`,`*n*`]`	Read at most *m*`*`*n* elements in column order. *n* can be `inf`, but *m* cannot.

When the *format* includes `%s`, *A* can contain more than *n* columns. *n* refers to elements, not characters.

str

Character string.

Output Arguments

A

An array. If the *format* includes:

- Only numeric specifiers, *A* is numeric. If *format* includes only 64-bit signed integer specifiers, *A* is of class `int64`. Similarly, if *format* includes only 64-bit unsigned integer specifiers, *A* is of class `uint64`. Otherwise, *A* is of class `double`. If *sizeA* is `inf` or *n*, then *A* is a column vector. If the input contains fewer than *sizeA* elements, MATLAB pads *A* with zeros.
- Only character or string specifiers (`%c` or `%s`), *A* is a character array. If *sizeA* is `inf` or *n*, *A* is a row vector. If the input contains fewer than *sizeA* characters, MATLAB pads *A* with `char(0)`.

- A combination of numeric and character specifiers, *A* is numeric, of class `double`. MATLAB converts each character to its numeric equivalent. This conversion occurs even when the *format* explicitly skips all numeric values (for example, a *format* of `'%*d %s'`).

If MATLAB cannot match the input to the *format*, and the *format* contains both numeric and character specifiers, *A* can be numeric or character. The class of *A* depends on the values MATLAB reads before processing stops.

count

Number of elements `sscanf` reads into *A*.

errmsg

An error message string when `sscanf` cannot open the specified file. Otherwise, an empty string.

nextindex

`sscanf` counts the number of characters `sscanf` reads from *str*, and then adds one.

Examples

Example 1

Read multiple floating-point values from a string:

```
s = '2.7183  3.1416';
A = sscanf(s,'%f')
A =
    2.7183
    3.1416
```

Example 2

Read an octal integer from a string, identified by the `'0'` prefix, using `%i` to preserve the sign:

```
sscanf('-010','%i')
ans =
```

```
-8
```

Example 3

Read numeric values from a two-dimensional character array. By default, `sscanf` reads characters in column order. To preserve the original order of the values, read one row at a time.

```
mixed = ['abc 45 6 ghi'; 'def 7 89 jkl'];

[nrows, ncols] = size(mixed);
for k = 1:nrows
    nums(k,:) = sscanf(mixed(k,:), '%*s %d %d %*s', [1, inf]);
end;

% type the variable name to see the result
nums =
    45     6
     7    89
```

Example 4

`sscanf` finds one match for `%s`

```
[str count] = sscanf('ThisIsOneString', '%s')
str =
    ThisIsOneString
count =
     1
```

`sscanf` finds four matches for `%s`. Because it does not match space characters, there are no spaces in the output string:

```
[str count] = sscanf('These Are Four Strings', '%s')
str =
    TheseAreFourStrings
count =
     4
```

`sscanf` finds five word matches for `%s` and four space character matches for `%c`. Because the `%c` specifier does match a space character, the output string does include spaces:

```
[str count] = sscanf('Five strings and four spaces', '%s%c')
str =
    Five strings and four spaces
count =
    9
```

`sscanf` finds three word matches for `%s` and two numeric matches for `%d`. Because the format specifier has a mixed %d and %s format, sscanf converts all nonnumeric characters to numeric:

```
[str count] = sscanf('5 strings and 4 spaces', '%d%s%s%d%s');
str'
  Columns 1 through 9
     5   115   116   114   105   110   103   115    97
  Columns 10 through 18
   110   100     4   115   112    97    99   101   115
count
count =
     5
```

Example 5

```
[str, count] = sscanf('one two three', '%c')
str =
    one two three
count =
    13

[str, count] = sscanf('one two three', '%13c')
str =
    one two three
count =
     1
```

```
[str, count] = sscanf('one two three', '%s')
str =
    onetwothree
count =
     3

[str, count] = sscanf('one two three', '%1s')
str =
    onetwothree
count =
    11
```

Example 6

```
tempString = '78 F 72 F 64 F 66 F 49 F';

degrees = char(176);
tempNumeric = sscanf(tempString, ['%d' degrees 'F'])'
tempNumeric =
    78    72    64    66    49
```

See Also

fscanf | sprintf | textscan

stairs

Purpose

Stairstep graph

Syntax

```
stairs(Y)
stairs(X,Y)
stairs(___,LineSpec)
stairs(___,Name,Value)

stairs(axes_handle,___)

h = stairs(___)

[xb,yb] = stairs(___)
```

Description

`stairs(Y)` draws a stairstep graph of the elements in `Y`.

- If `Y` is a vector, then the *x*-axis scale ranges from 1 to `length(Y)`.
- If `Y` is a matrix, then `stairs` draws one line per matrix column and the *x*-axis scale ranges from 1 to the number of rows in `Y`.

`stairs(X,Y)` plots the elements in `Y` at the locations specified in `X`. The inputs `X` and `Y` must be vectors or matrices of the same size. Additionally, `X` can be a row or column vector and `Y` must be a matrix with `length(X)` rows.

`stairs(___,LineSpec)` specifies a line style, marker symbol, and color. Use this option with any of the input argument combinations in the previous syntaxes.

`stairs(___,Name,Value)` specifies stairseries properties using one or more `Name,Value` pair arguments.

`stairs(axes_handle,___)` plots into the axes specified by `axes_handle` instead of into the current axes (`gca`). The option, `axes_handle`, can precede any of the input argument combinations in the previous syntaxes.

`h = stairs(___)` returns a vector of stairseries object handles in `h`. When multiple stairseries are present, you can make changes to properties of a specific stairseries by specifying a particular handle.

`[xb,yb] = stairs(___)` does not create a plot, but returns matrices `xb` and `yb` of the same size, such that `plot(xb,yb)` plots the stairstep graph.

Input Arguments

Y - Elements to plot

vector or matrix

Elements to plot, specified as a vector or matrix. When `Y` is a vector, `stairs` creates one stairseries. When `Y` is a matrix, `stairs` draws one line per matrix column and creates a separate stairseries for each column.

Data Types

`single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical`

X - Locations to plot elements in Y

vector or matrix

Locations to plot elements in `Y`, specified as a vector or matrix. When `Y` is a vector, `X` must be a vector of the same size. When `Y` is a matrix, `X` must be a matrix of the same size, or a vector whose length equals the number of rows in `Y`.

Data Types

`single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical`

LineSpec - Line style, marker symbol, and color

string

Line style, marker symbol, and color, specified as a string. For more information on line style, marker symbol, and color options see `LineSpec`.

Example: `':*r'`

Data Types
`char`

axes_handle - Axes handle

Axes handle, which is the reference to an axes object. Use the `gca` function to get the handle to the current axes, for example, `axes_handle = gca;`.

Name-Value Pair Arguments

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

Example: `'Marker','s','MarkerFaceColor','red'` plots the stairstep graph with red square markers.

For more information on stairseries properties see `stairseries`.

'LineStyle' - Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as the comma-separated pair consisting of `'LineStyle'` and a line style specifier. This table lists supported line styles.

Specifier	Line Style
'- '	Solid line (default)
'--'	Dashed line
':'	Dotted line

Specifier	Line Style
'-.'	Dash-dot line
'none'	No line

Example: 'LineStyle','-.'

'LineWidth' - Line width

0.5 (default) | scalar

Line width, specified as the comma-separated pair consisting of 'LineWidth' and a scalar in points.

Example: 'LineWidth',0.75

'Color' - Color

[0 0 1] (blue) (default) | 3-element RGB vector | string

Color, specified as the comma-separated pair consisting of 'Color' and a 3-element RGB vector or a string containing the short or long name of the color. The RGB vector is a 3-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0 1].

This table lists the predefined colors and their RGB equivalents.

RGB Vector	Short Name	Long Name
[1 1 0]	'y'	'yellow'
[1 0 1]	'm'	'magenta'
[0 1 1]	'c'	'cyan'
[1 0 0]	'r'	'red'
[0 1 0]	'g'	'green'
[0 0 1]	'b'	'blue'
[1 1 1]	'w'	'white'
[0 0 0]	'k'	'black'

Example: `'Color',[0 1 0]`

Example: `'Color','green'`

'Marker' - Marker symbol

`'none'` (default) | string

Marker symbol, specified as the comma-separated pair consisting of `'Marker'` and a marker specifier. This table lists supported marker symbols.

Specifier	MarkerSymbol
'o'	Circle
'+'	Plus sign
'*'	Asterisk
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No marker

Example: `'Marker','+'`

Example: `'Marker','diamond'`

'MarkerEdgeColor' - Marker edge color

`[0 0 1]` (blue) (default) | `'auto'` | `'none'` | three-element RGB vector | string

Marker edge color, specified as the comma-separated pair consisting of `'MarkerEdgeColor'` and a color value. The color value can be one of the supported strings or an RGB vector, listed in the following tables.

Specifier	Result
'auto'	Uses same color as line color
'none'	Specifies no color, which makes unfilled markers invisible

For an RGB vector, use a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0 1]`. This table lists the predefined string colors and their RGB equivalents.

RGB Vector	Short Name	Long Name
[1 1 0]	'y'	'yellow'
[1 0 1]	'm'	'magenta'
[0 1 1]	'c'	'cyan'
[1 0 0]	'r'	'red'
[0 1 0]	'g'	'green'
[0 0 1]	'b'	'blue'
[1 1 1]	'w'	'white'
[0 0 0]	'k'	'black'

Example: `'MarkerEdgeColor',[1 .8 .1]`

'MarkerFaceColor' - Marker face color

`'none'` (default) | `'auto'` | three-element RGB vector | string

Marker face color, specified as the comma-separated pair consisting of `'MarkerFaceColor'` and a color value. `MarkerFaceColor` sets the fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). The color value can be one of the supported strings or an RGB vector, listed in the following tables.

Specifier	Result
'auto'	Uses same color as marker edge color
'none'	Makes the interior of the marker transparent, allowing the background to show through (default)

For an RGB vector, use a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0 1]`. This table lists the predefined string colors and their RGB equivalents.

RGB Vector	Short Name	Long Name
[1 1 0]	'y'	'yellow'
[1 0 1]	'm'	'magenta'
[0 1 1]	'c'	'cyan'
[1 0 0]	'r'	'red'
[0 1 0]	'g'	'green'
[0 0 1]	'b'	'blue'
[1 1 1]	'w'	'white'
[0 0 0]	'k'	'black'

Example: `'MarkerFaceColor',[0 .8 1]`

'MarkerSize' - Marker size

6 (default) | scalar

Marker size, specified as the comma-separated pair consisting of 'MarkerSize' and a scalar in points.

Example: 'MarkerSize',7

Output Arguments

h - Stairseries object handle

scalar or column vector

Stairseries object handle, specified as a scalar or column vector. This is a unique identifier, which you can use to query and modify the properties of a specific stairseries.

xb - x values for use with plot

vector or matrix

x values for use with plot, specified as a vector or matrix. xb contains the appropriate values such that plot(xb,yb) creates the stairstep graph.

yb - y values for use with plot

vector or matrix

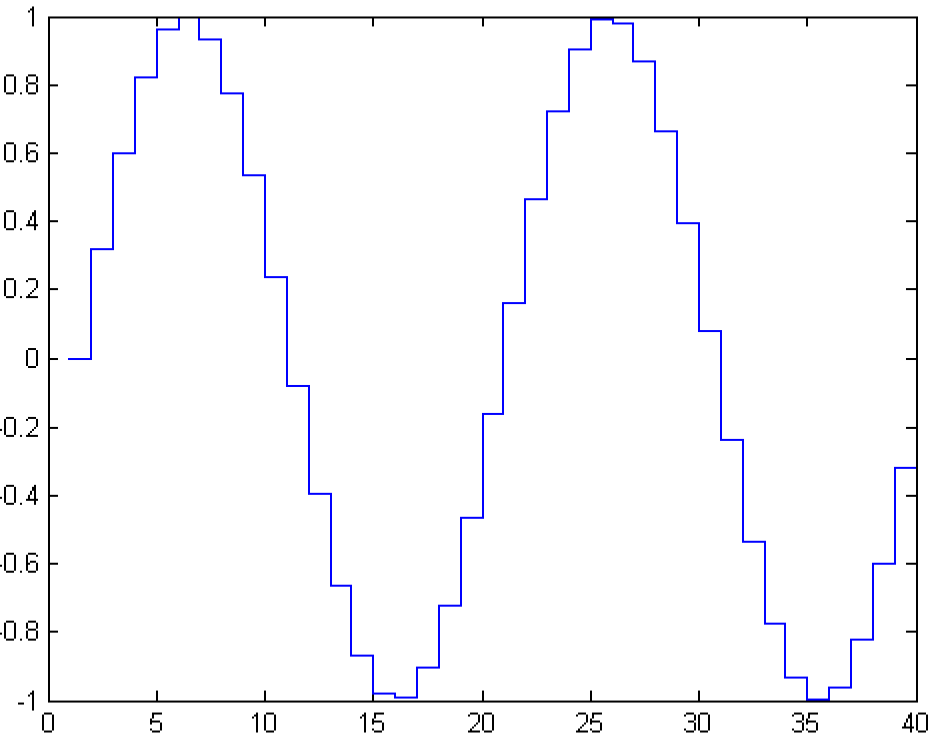
y values for use with plot, specified as a vector or matrix. yb contains the appropriate values such that plot(xb,yb) creates the stairstep graph.

Examples

Plot a Single Data Series

Create a stairstep plot of sine evaluated at 40 equally spaced values between 0 and 4π.

```
figure
X = linspace(0,4*pi,40);
Y = sin(X);
stairs(Y)
```

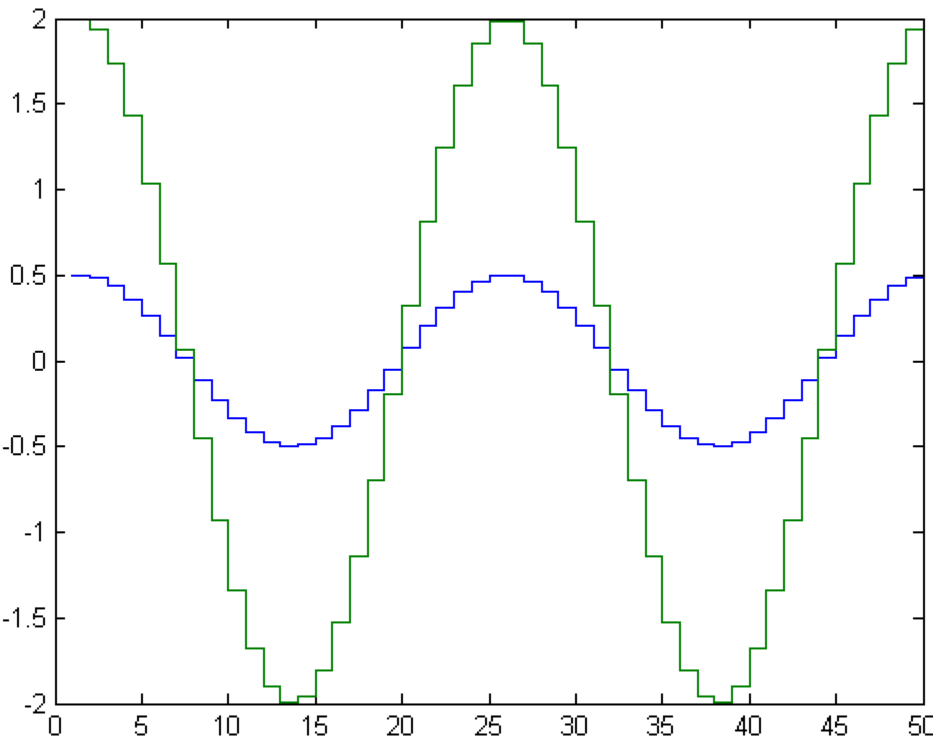


The length of `Y` automatically determines and generates the *x*-axis scale.

Plot Multiple Data Series

Create a stairstep plot of two cosine functions evaluated at 50 equally spaced values between 0 and 4π.

```
figure
X = linspace(0,4*pi,50)';
Y = [0.5*cos(X), 2*cos(X)];
stairs(Y)
```



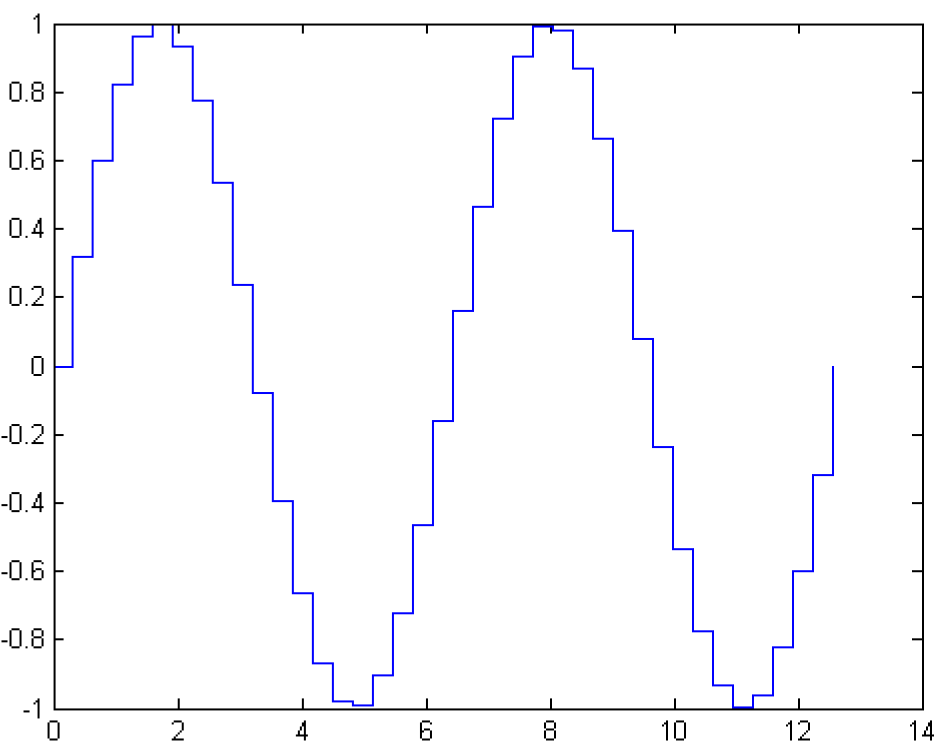
The number of rows in Y automatically determines and generates the *x*-axis scale.

Plot a Single Data Series at Specified *x* Values

Create a stairstep plot of a sine wave evaluated at equally spaced values between 0 and 4π. Specify the set of *x* values for the plot.

```
figure
X = linspace(0,4*pi,40);
Y = sin(X);
```

```
stairs(X,Y)
```



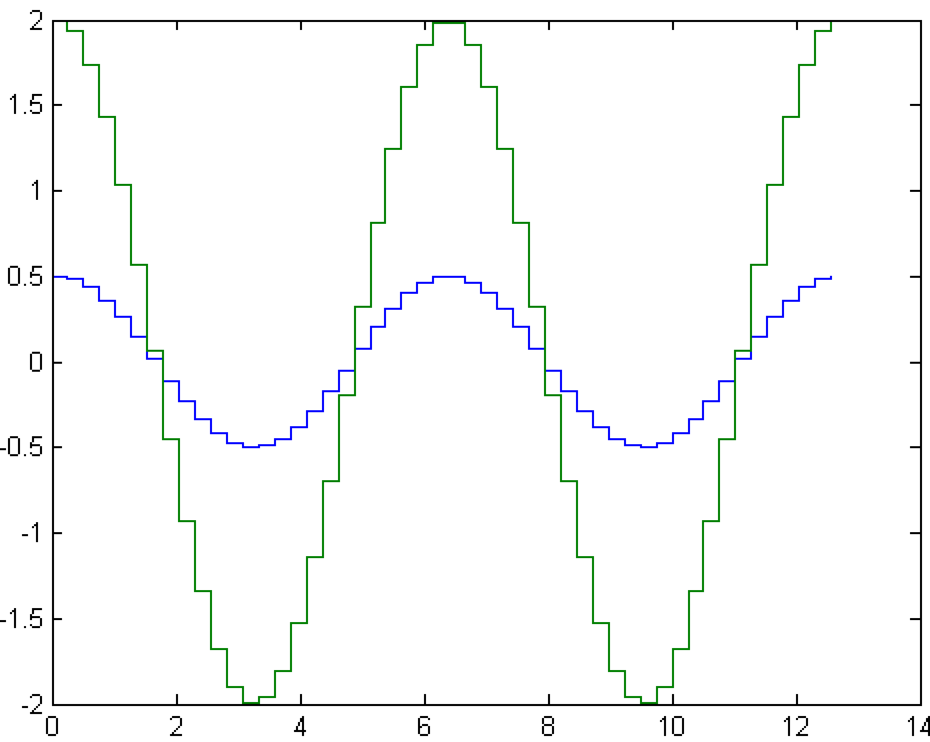
The entries in Y are plotted against the corresponding entries in X.

Plot Multiple Data Series at Specified *x* Values

Create a stairstep plot of two cosine waves evaluated at equally spaced values between 0 and 4π. Specify the set of *x* values for the plot.

```
figure
X = linspace(0,4*pi,50)';
```

```
Y = [0.5*cos(X), 2*cos(X)];
stairs(X,Y)
```

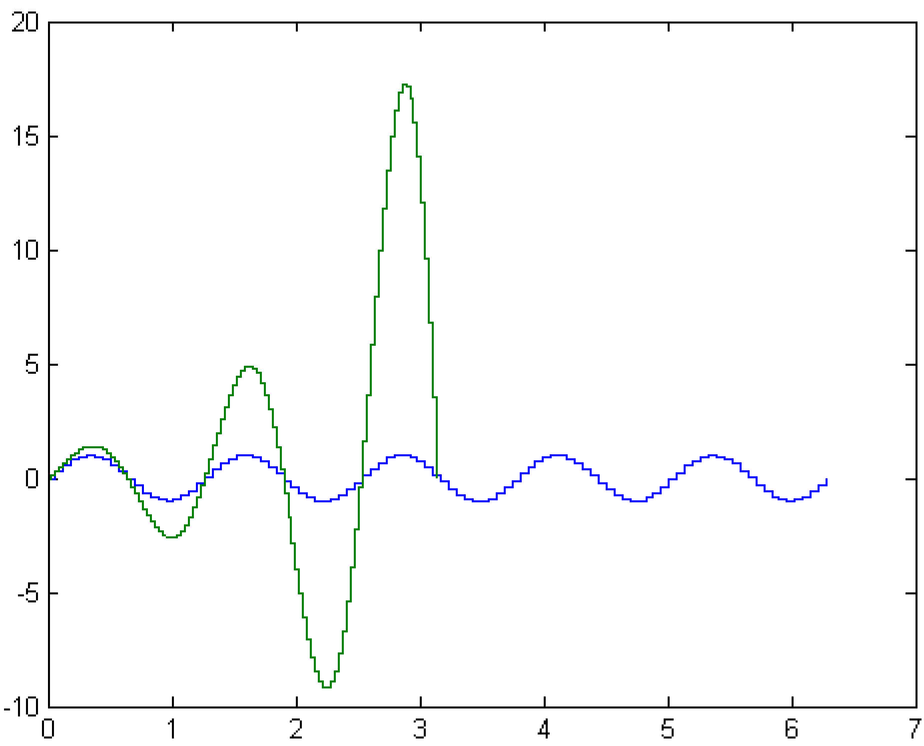


The first vector input, X, determines the *x*-axis positions for both data series.

Plot Multiple Data Series at Unique Sets of *x* Values

Create a stairstep plot of two sine waves evaluated at different values. Specify a unique set of *x* values for plotting each data series.

```
figure
x1 = linspace(0,2*pi)';
x2 = linspace(0,pi)';
X = [x1,x2];
Y = [sin(5*x1),exp(x2).*sin(5*x2)];
stairs(X,Y)
```

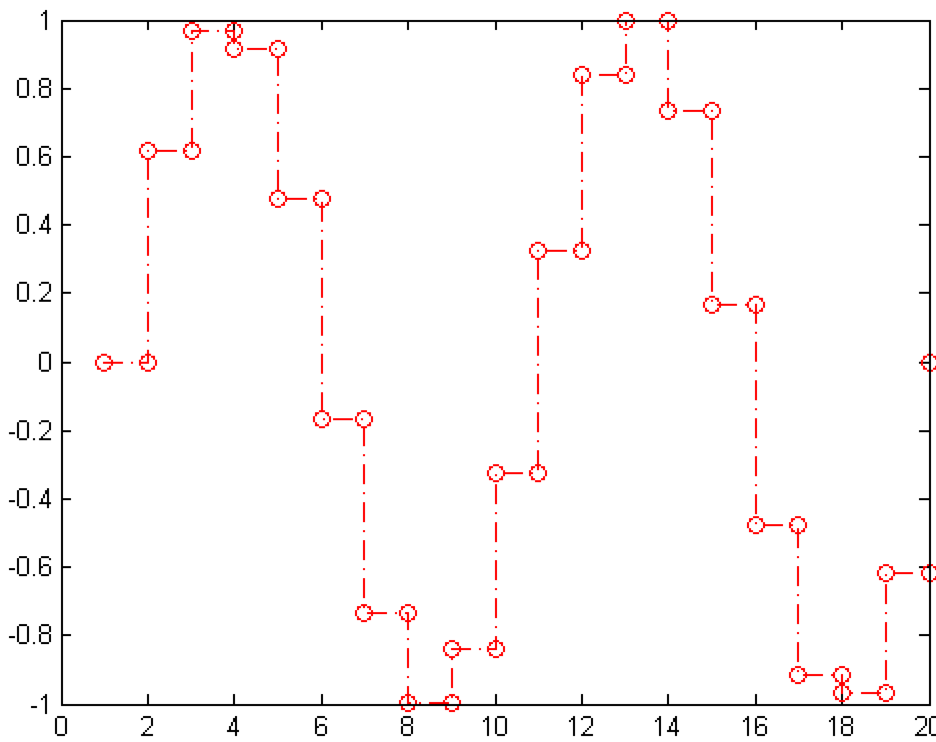


Each column of X is plotted against the corresponding column of Y.

Specify the Line Style, Marker Symbol and Color

Create a stairstep plot and set the line style to a dot-dashed line, the marker symbol to circles, and the color to red.

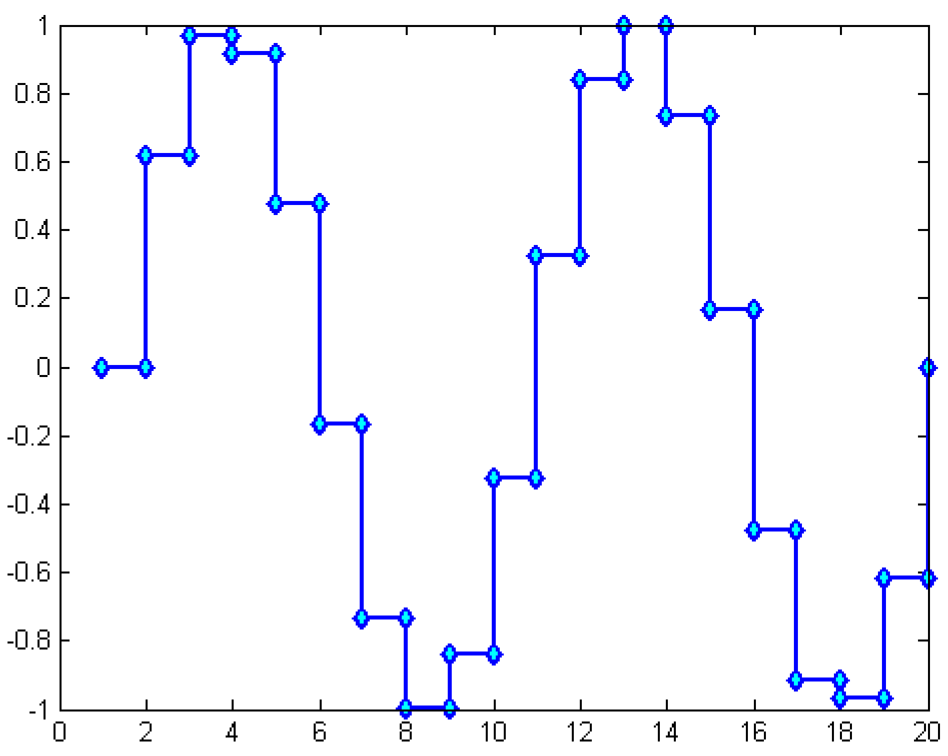
```
figure
X = linspace(0,4*pi,20);
Y = sin(X);
stairs(Y, '-.or')
```



Specify Additional Style Options

Create a stairstep plot and set the line width to 2 points, the marker symbols to diamonds, and the marker face color to cyan using `Name,Value` pair arguments.

```
figure
X = linspace(0,4*pi,20);
Y = sin(X);
stairs(Y,'LineWidth',2,'Marker','d','MarkerFaceColor','c')
```



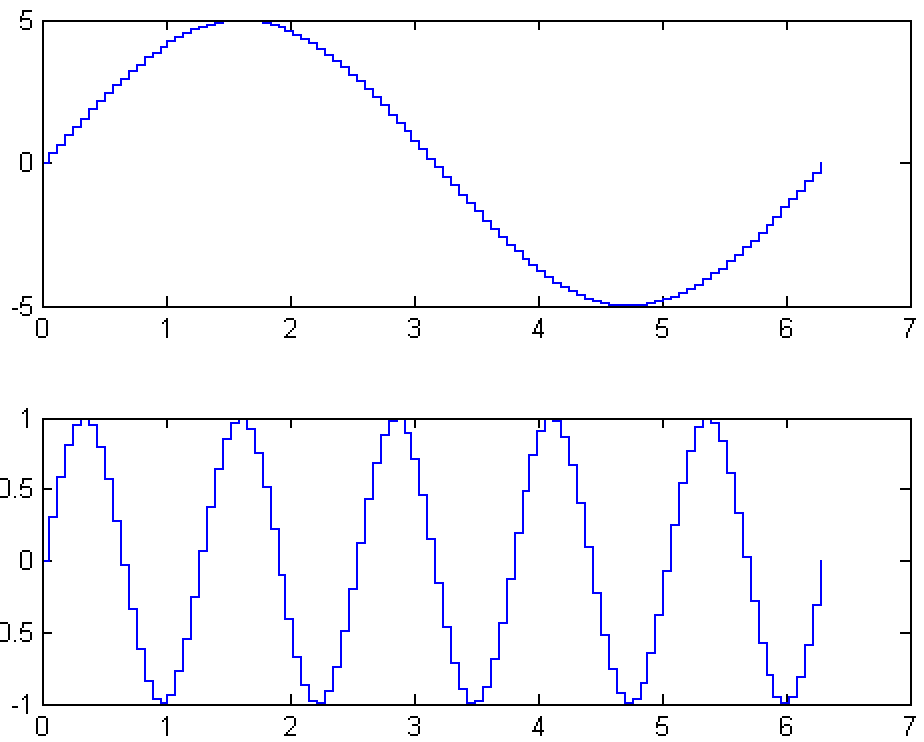
Specify Axes for Stairstep Plots

Create two subplots and return the handles to the axes, `s(1)` and `s(2)`.

```
figure
s(1) = subplot(2,1,1);
s(2) = subplot(2,1,2);
```

Create a stairstep plot in each subplot by referring to the axes handles.

```
X = linspace(0,2*pi);
Y1 = 5*sin(X);
Y2 = sin(5*X);
stairs(s(1),X,Y1)
stairs(s(2),X,Y2)
```



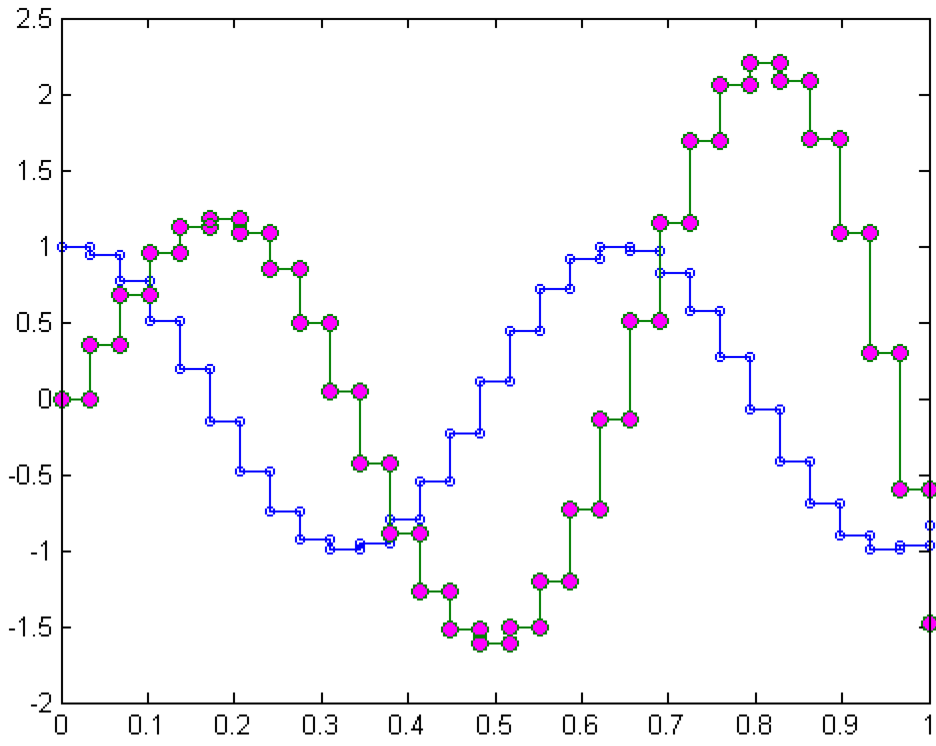
Return Stairseries Handle

Create a stairstep plot of two data series and return the handle to each stairseries.

```
figure
X = linspace(0,1,30)';
Y = [cos(10*X), exp(X).*sin(10*X)];
h = stairs(X,Y);
```

Specify the markers of the first series to small circles and the second series to magenta filled circles by referring to the handles.

```
set(h(1),'Marker','o','MarkerSize',4)
set(h(2),'Marker','o','MarkerFaceColor','m')
```



Create a Stairstep Plot using `plot`

Evaluate two cosine functions at 40 equally spaced values between 0 and 4π and create a stairstep plot using `plot`.

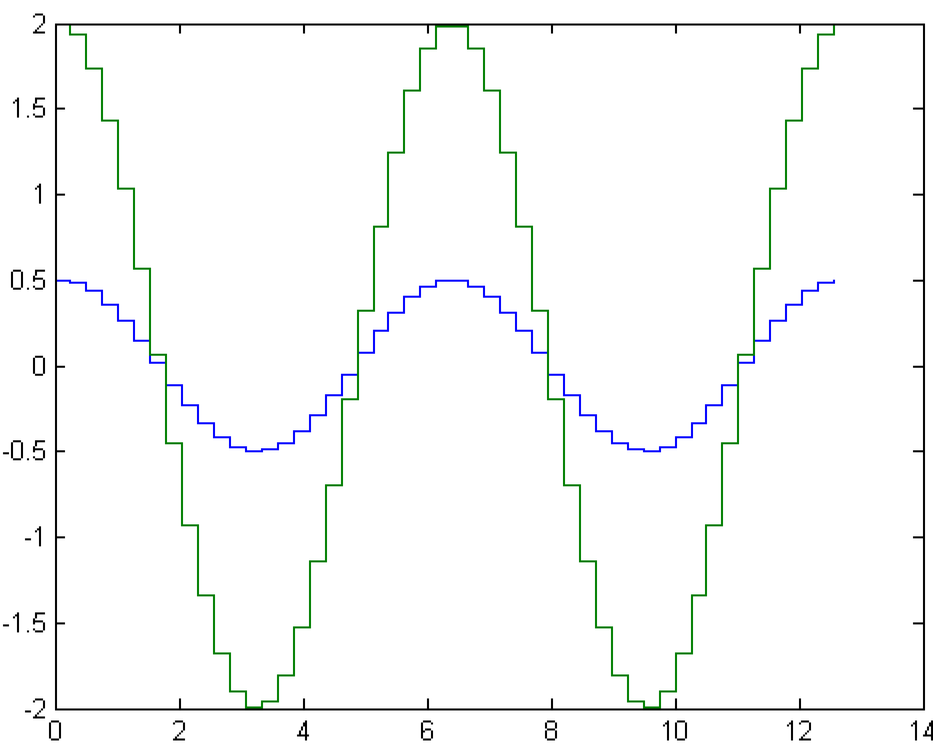
Use `stairs` to get `xb` and `yb`.

```
figure
X = linspace(0,4*pi,50)';
Y = [0.5*cos(X), 2*cos(X)];
[xb,yb] = stairs(X,Y);
```

`stairs` returns two matrices of the same size, but no plot.

Use `plot` to create the stairstep plot with `xb` and `yb`.

```
plot(xb,yb)
```



See Also

`bar` | `hist` | `stem` | `LineSpec` `Stairseries`

Stairseries Properties

Purpose

Define stairseries properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands or the Property Editor (`propertyeditor`).

Note that you cannot define default property values for stairseries objects.

See Plot Objects for information on stairseries objects.

Stairseries Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`Annotation`

`hg.Annotation` object (read-only)

Control the display of stairseries objects in legends. Specifies whether this stairseries object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the stairseries object is displayed in a figure legend.

IconDisplayStyle Value	Purpose
`on`	Include the stairseries object in a legend as one entry, but not its children objects
`off`	Do not include the stairseries or its children in a legend (default)
`children`	Include only the children of the stairseries as separate entries in the legend

Setting the IconDisplayStyle Property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the IconDisplayStyle Property

See "Controlling Legends" for more information and examples.

`BeingDeleted`

`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the

running callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`

string | function handle

Button press callback function. Executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be:

- A string that is a valid MATLAB expression
- The name of a MATLAB file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

The expression executes in the MATLAB workspace.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`Children`

array of graphics object handles

Children of the stairseries object. An array containing the handles of all line objects parented to the stairseries object (whether visible or not).

If a child object's `HandleVisibility` property is `callback` or `off`, its handle does not show up in this object's `Children` property. If you want the handle in the `Children` property, set the root `ShowHiddenHandles` property to `on`. For example:

```
set(0,'ShowHiddenHandles','on')
```

`Clipping`

`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs appear outside the axes plot box. This occurs if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`Color`

ColorSpec

Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object's color. The default value is `[0 0 0]` (`black`).

See the `ColorSpec` reference page for more information on specifying color. See "Adding Arrows and Lines to Graphs".

`CreateFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback function executed during object creation. Defines a callback that executes when MATLAB creates an object. The default is an empty array.

You must specify the callback during the creation of the object. For example:

```
graphicfcn(y,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function and `graphicfcn` is the plotting function which creates this object.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DeleteFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback executed during object deletion. Executes when this object is deleted (for example, this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values. The default is an empty array.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`
string

String used by legend. The `legend` function uses the `DisplayName` property to label the stairseries object in the legend. The default is an empty string.

- If you specify string arguments with the `legend` function, MATLAB set `DisplayName` to the corresponding string and uses that string for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, MATLAB set `DisplayName` to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add a legend programmatically that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more information and examples.

`EraseMode`
`{normal} | none | xor | background`

Erase mode. Controls the technique MATLAB uses to draw and erase objects. Use alternative erase modes for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the object when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it because MATLAB stores no information about its former location.
- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object's color depends on the color of whatever is beneath it on the display.
- `background` — Erase the object by redrawing it in the axes background `Color`, or the figure background `Color` if the axes `Color` property is `none`. This damages objects that are behind the erased object, but properly colors the erased object.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors (for example, performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting

to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible.
- `callback` — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property,

figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties. See also `findall`.

Handle Validity

Hidden handles are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`
`{on} | off`

Selectable by mouse click. Determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the line objects that compose the stairstep graph. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`HitTestArea`
`on | {off}`

Select the object by clicking lines or area of extent. Select plot objects by:

- Clicking lines (default).
- Clicking anywhere in the extent of the plot.

When `HitTestArea` is `off`, you must click the lines to select the object. When `HitTestArea` is `on`, you can select this object by clicking anywhere within the extent of the plot (that is, anywhere within a rectangle that encloses the stairstep graph).

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the `Interruptible` property affects only the callbacks for the`ButtonDownFcn` property. A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both the callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.

- If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see "Control Callback Execution and Interruption".

`LineStyle`

`{-} | -- | :  | -.  | none`

Line style of stairseries object.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line

Specifier	Line Style
'-.'	Dash-dot line
'none'	No line

Use `LineStyle` `none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`
size in points

Width of linear objects and edges of filled areas. Specify in points. 1 point = $^1/_{72}$ inch. The default is 0.5 points.

`Marker`
character (see table)

Marker symbol. Specifies marks that display at data points. You can set values for the `Marker` property independently from the `LineStyle` property. For a list of supported marker symbols, see the following table.

Marker Specifiers Table

Specifier	Marker Type
'+'	Plus sign
'o'	Circle
'*'	Asterisk
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond

Specifier	Marker Type
’^’	Upward-pointing triangle
’v’	Downward-pointing triangle
’>’	Right-pointing triangle
’<’	Left-pointing triangle
’pentagram’ or ’p’	Five-pointed star (pentagram)
’hexagram’ or ’h''’	Six-pointed star (hexagram)
’none’	No marker (default)

MarkerEdgeColor
ColorSpec | none | {auto}

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- ColorSpec — User-defined color.
- none — Specifies no color, which makes nonfilled markers invisible.
- auto — Uses same color as the Color property.

MarkerFaceColor
ColorSpec | {none} | auto

Fill color for closed-shape markers. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- ColorSpec — User-defined color.
- none — Makes the interior of the marker transparent, allowing the background to show through.
- auto — Sets the fill color to the axes Color property. If the axes Color property is none, sets the fill color to the figure Color.

`MarkerSize`
scalar

Marker size. Size of the marker in points. The default value is `6`.

Note MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`Parent`
handle of parent axes, hggroup, or hgtransform

Parent of object. Handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Object selection state. When you set this property to `on`, MATLAB displays selection handles at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
`{on} | off`

Object highlighted when selected.

- `on` — MATLAB indicates the selected state by drawing four edge handles and four corner handles.

- `off` — MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`Tag`
string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

For example, create an areaseries object and set the `Tag` property.

```
t = area(Y,'Tag','area1')
```

When you want to access objects of a given type, use `findobj` to find the object's handle. The following statement changes the `FaceColor` property of the object whose `Tag` is `area1`.

```
set(findobj('Tag','area1'),'FaceColor','red')
```

`Type`
string (read-only)

Type of graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For stairseries objects, `Type` is '`hggroup`'. The following statement finds all the hggroup objects in the current axes object.

```
t = findobj(gca,'Type','hggroup');
```

`UIContextMenu`
handle of uicontextmenu object

Associate context menu with object. Handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu`

function to create the context menu. MATLAB displays the context menu whenever you right-click over the object. The default value is an empty array.

`UserData`
array

User-specified data. Data you want to associate with this object (including cell arrays and structures). The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`
`{on} | off`

Visibility of object and its children.

- `on` — Object and all children of the object are visible unless the child object's `Visible` property is `off`.
- `off` — Object not displayed. However, the object still exists and you can set and query its properties.

`XData`
array

X-axis location of stairs. The `stairs` function uses `XData` to label the *x*-axis. `XData` can be either a matrix equal in size to `YData` or a vector equal in length to the number of rows in `YData`. That is, `length(XData) == size(YData,1)`.

If you do not specify `XData` (which the input argument `X`), the `stairs` function uses the indices of `YData` to create the stairstep graph. See the `XDataMode` property for related information.

`XDataMode`
`{auto} | manual`

Use automatic or user-specified x-axis values. If you specify `XData` (by setting the `XData` property or specifying the `X` input

argument), MATLAB sets this property to `manual` and uses the specified values to label the *x*-axis.

If you set `XDataMode` to `auto` after specifying `XData`, MATLAB resets the *x*-axis ticks to `1:size(YData,1)` or to the column indices of the `ZData`, overwriting any previous values for `XData`.

`XDataSource`

MATLAB variable, as a string

Link `XData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `XData`. The default value is an empty array.

```
set(h,'XDataSource','xdatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `XDataSource` does not change the object's `XData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`YData`

scalar | vector | matrix

Stairs plot data. `YData` contains the data plotted in the stairstep graph. Each value in `YData` is represented by a marker in the stairstep graph. If `YData` is a matrix, the `stairs` function creates a line for each column in the matrix.

The input argument `Y` in the `stairs` function calling syntax assigns values to `YData`.

`YDataSource`

MATLAB variable, as a string

Link `YData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `YData`. The default value is an empty array.

```
set(h,'YDataSource','Ydatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `YDataSource` does not change the object's `YData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Purpose

Start timer(s) running

Syntax

```
start(obj)
```

Description

`start(obj)` starts the timer running, represented by the timer object, `obj`. If `obj` is an array of timer objects, `start` starts all the timers. Use the `timer` function to create a timer object.

`start` sets the `Running` property of the timer object, `obj`, to `'on'`, initiates `TimerFcn` callbacks, and executes the `StartFcn` callback.

The timer stops running if one of the following conditions apply:

- The first `TimerFcn` callback completes, if `ExecutionMode` is `'singleShot'`.
- The number of `TimerFcn` callbacks specified in `TasksToExecute` have been executed.
- The `stop(obj)` command is issued.
- An error occurred while executing a `TimerFcn` callback.

See Also

`timer` | `stop`

Purpose

Start timer(s) running at specified time

Syntax

```
startat(obj,time)
startat(obj,S)
startat(obj,S,pivotyear)
startat(obj,Y,M,D)
startat(obj,[Y,M,D])
startat(obj,Y,M,D,H,MI,S)
startat(obj,[Y,M,D,H,MI,S])
```

Description

`startat(obj,time)` starts the timer represented by timer object `obj`running at the time specified by the serial date number `time`. If `obj` is an array of timer objects, `startat` starts all the timers running at the specified time. To create a timer object, use the `timer` function. You can set the starting *`time`* to any serial date number less than or equal to 25 days from the current date.

`startat` sets the `Running` property of the timer object, `obj`, to `'on'`, initiates `TimerFcn` callbacks, and executes the `StartFcn` callback.

The serial date number, `time`, indicates the number of days that have elapsed since 1-Jan-0000 (starting at 1). See `datenum` for additional information about serial date numbers.

`startat(obj,S)` starts the timer running at the time specified by the date string `S`. The date string must use date format 0, 1, 2, 6, 13, 14, 15, 16, or 23, as defined by the `datestr` function. Date strings with two-character years are interpreted to be within the 100 years centered on the current year.

`startat(obj,S,pivotyear)` uses the specified pivot year as the starting year of the 100-year range in which a two-character year resides. The default pivot year is the current year minus 50 years.

`startat(obj,Y,M,D)` `startat(obj,[Y,M,D])` start the timer at the year (`Y`), month (`M`), and day (`D`) specified. `Y`, `M`, and `D` must be arrays of the same size (or they can be a scalar).

`startat(obj,Y,M,D,H,MI,S)` `startat(obj,[Y,M,D,H,MI,S])` start the timer at the year (`Y`), month (`M`), day (`D`), hour (`H`), minute (`MI`), and

second (S) specified. Y, M, D, H, MI, and S must be arrays of the same size (or they can be a scalar). Values outside the normal range of each array are automatically carried to the next unit (for example, month values greater than 12 are carried to years). Month values less than 1 are set to be 1; all other units can wrap and have valid negative values.

The timer stops running if one of the following conditions apply:

- The number of TimerFcn callbacks specified in TasksToExecute have been executed.
- The stop(obj) command is issued.
- An error occurred while executing a TimerFcn callback.

Examples

This example uses a timer object to execute a function at a specified time.

```
t1=timer('TimerFcn','disp(''it is 10 o''''clock'')');
startat(t1,'10:00:00');
```

This example uses a timer to display a message when an hour has elapsed.

```
t2=timer('TimerFcn','disp(''It has been an hour now.'')');
startat(t2,now+1/24);
```

See Also

datenum | datestr | now | timer | start | stop

Purpose

Startup file for user-defined options

Syntax

```
startup
```

Description

`startup` executes commands of your choosing when the MATLAB program starts.

Create a `startup.m` file in your MATLAB startup folder and put in the file any commands you want executed at MATLAB startup. For example, your `startup.m` file might include physical constants, defaults for Handle Graphics properties, engineering conversion factors, or anything else you want predefined in your workspace.

Algorithms

The MATLAB program executes the `matlabrc.m` file when it starts. `matlabrc.m` invokes `startup.m`, if it exists on the MATLAB search path.

You can extend this process to create additional startup files, if needed.

MathWorks does not recommend modifying the `matlabrc.m` file, except perhaps by system administrators in network configurations.

See Also

`finish` | `matlabrc` | `matlabroot` | `path` | `quit` | `userpath`

How To

- "Specifying Startup Options in the MATLAB Startup File"
- Preferences

Purpose

Standard deviation

Syntax

```
s = std(X)
s = std(X,flag)
s = std(X,flag,dim)
```

Definitions

There are two common textbook definitions for the standard deviation `s` of a data vector `X`.

$$(1)\ s = \left(\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \right)^{\frac{1}{2}}$$

$$(2)\ s = \left(\frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2 \right)^{\frac{1}{2}},$$

where

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

and n is the number of elements in the sample. The two forms of the equation differ only in $n-1$ versus n in the divisor.

Description

`s = std(X)`, where `X` is a vector, returns the standard deviation using (1) above. The result `s` is the square root of an unbiased estimator of the variance of the population from which `X` is drawn, as long as `X` consists of independent, identically distributed samples.

If `X` is a matrix, `std(X)` returns a row vector containing the standard deviation of the elements of each column of `X`. If `X` is a multidimensional array, `std(X)` is the standard deviation of the elements along the first nonsingleton dimension of `X`.

`s = std(X,flag)` for `flag = 0`, is the same as `std(X)`. For `flag = 1`, `std(X,1)` returns the standard deviation using (2) above, producing the second moment of the set of values about their mean.

`s = std(X,flag,dim)` computes the standard deviations along the dimension of `X` specified by scalar `dim`. Set `flag` to `0` to normalize `Y` by *n*-1; set `flag` to `1` to normalize by *n*.

The input array, `X`, must be of type `double` or `single` for all syntaxes.

Examples

For matrix `X`

```
X =
      1     5     9
     7    15    22
s = std(X,0,1)
s =
    4.2426    7.0711   9.1924
s = std(X,0,2)
s =
    4.000
    7.5056
```

See Also

`corrcoef` | `cov` | `mean` | `median` | `var`

Purpose

Plot discrete sequence data

Syntax

```
stem(Y)
stem(X,Y)
stem(___,'fill')
stem(___,LineSpec)
stem(___,Name,Value)

stem(axes_handle,___)

h = stem(___)
```

Description

`stem(Y)` plots the data sequence, `Y`, as stems that extend from a baseline along the *x*-axis. The data values are indicated by circles terminating each stem.

- If `Y` is a vector, then the *x*-axis scale ranges from 1 to `length(Y)`.
- If `Y` is a matrix, then `stem` plots all elements in a row against the same *x* value, and the *x*-axis scale ranges from 1 to the number of rows in `Y`.

`stem(X,Y)` plots the data sequence, `Y`, at values specified by `X`. The `X` and `Y` inputs must be vectors or matrices of the same size. Additionally, `X` can be a row or column vector and `Y` must be a matrix with `length(X)` rows.

- If `X` and `Y` are both vectors, then `stem` plots entries in `Y` against corresponding entries in `X`.
- If `X` is a vector and `Y` is a matrix, then `stem` plots each column of `Y` against the set of values specified by `X`, such that all elements in a row of `Y` are plotted against the same value.
- If `X` and `Y` are both matrices, then `stem` plots columns of `Y` against corresponding columns of `X`.

`stem(___,'fill')` fills the circles. Use this option with any of the input argument combinations in the previous syntaxes.

`stem(___,LineSpec)` specifies the line style, marker symbol, and color.

`stem(___,Name,Value)` specifies stemseries properties using one or more `Name,Value` pair arguments.

`stem(axes_handle,___)` plots into the axes specified by `axes_handle` instead of into the current axes (`gca`). The option, `axes_handle`, can precede any of the input argument combinations in the previous syntaxes.

`h = stem(___)` returns a vector of stemseries object handles in `h`. When multiple stemseries objects are present, you can make changes to properties of a specific stemseries by specifying a particular handle.

Input Arguments

Y - Data sequence to display

vector or matrix

Data sequence to display, specified as a vector or matrix. When `Y` is a vector, `stem` creates one stemseries. When `Y` is a matrix, `stem` creates a separate stemseries for each column.

Data Types

`single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical`

X - Locations to plot data values in Y

vector or matrix

Locations to plot data values in `Y`, specified as a vector or matrix. When `Y` is a vector, `X` must be a vector of the same size. When `Y` is a matrix, `X` must be a matrix of the same size, or a vector whose length equals the number of rows in `Y`.

Data Types

`single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical`

LineSpec - Line style, marker symbol, and color

string

Line style, marker symbol, and color, specified as a string. For more information on line style, marker symbol, and color options see `LineSpec`.

Example: `':*r'`

Data Types
`char`

axes_handle - Axes handle

Axes handle, which is the reference to an axes object. Use the `gca` function to get the handle to the current axes, for example, `axes_handle = gca;`.

Name-Value Pair Arguments

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

Example: `'LineStyle',':','MarkerFaceColor','red'` plots the stem as a dotted line and colors the marker face red.

For more information on these properties see `stemseries`.

'LineStyle' - Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as the comma-separated pair consisting of `'LineStyle'` and a line style specifier. This table lists supported line styles.

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line
'none'	No line

Example: `'LineStyle','-.'`

'LineWidth' - Line width

`0.5` (default) | scalar

Line width, specified as the comma-separated pair consisting of `'LineWidth'` and a scalar. The scalar sets the width size in points of the stem and marker edge.

Example: `'LineWidth',0.75`

'Color' - Color

`[0 0 1]` (blue) (default) | 3-element RGB vector | string

Color, specified as the comma-separated pair consisting of `'Color'` and a 3-element RGB vector or a string containing the short or long name of the color. The RGB vector is a 3-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0 1]`.

This table lists the predefined colors and their RGB equivalents.

RGB Vector	Short Name	Long Name
[1 1 0]	'y'	'yellow'
[1 0 1]	'm'	'magenta'

RGB Vector	Short Name	Long Name
[0 1 1]	'c'	'cyan'
[1 0 0]	'r'	'red'
[0 1 0]	'g'	'green'
[0 0 1]	'b'	'blue'
[1 1 1]	'w'	'white'
[0 0 0]	'k'	'black'

Example: 'Color',[0 1 0]

Example: 'Color','green'

'Marker' - Marker symbol

'o' (default) | string

Marker symbol, specified as the comma-separated pair consisting of 'Marker' and a marker specifier. This table lists supported marker symbols.

Specifier	MarkerSymbol
'o'	Circle
'+'	Plus sign
'*'	Asterisk
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle

Specifier	MarkerSymbol
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No marker

Example: 'Marker','+'

Example: 'Marker','diamond'

'MarkerEdgeColor' - Marker edge color

[0 0 1] (blue) (default) | 'auto' | 'none' | three-element RGB vector | string

Marker edge color, specified as the comma-separated pair consisting of 'MarkerEdgeColor' and a color value. The color value can be one of the supported strings or an RGB vector, listed in the following tables.

Specifier	Result
'auto'	Uses same color as line color
'none'	Specifies no color, which makes unfilled markers invisible

For an RGB vector, use a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0 1]. This table lists the predefined string colors and their RGB equivalents.

RGB Vector	Short Name	Long Name
[1 1 0]	'y'	'yellow'
[1 0 1]	'm'	'magenta'

RGB Vector	Short Name	Long Name
[0 1 1]	'c'	'cyan'
[1 0 0]	'r'	'red'
[0 1 0]	'g'	'green'
[0 0 1]	'b'	'blue'
[1 1 1]	'w'	'white'
[0 0 0]	'k'	'black'

Example: `'MarkerEdgeColor',[1 .8 .1]`

'MarkerFaceColor' - Marker face color

`'none'` (default) | `'auto'` | three-element RGB vector | string

Marker face color, specified as the comma-separated pair consisting of `'MarkerFaceColor'` and a color value. `MarkerFaceColor` sets the fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). The color value can be one of the supported strings or an RGB vector, listed in the following tables.

Specifier	Result
'auto'	Uses same color as marker edge color
'none'	Makes the interior of the marker transparent, allowing the background to show through (default)

For an RGB vector, use a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0 1]`. This table lists the predefined string colors and their RGB equivalents.

RGB Vector	Short Name	Long Name
[1 1 0]	'y'	'yellow'
[1 0 1]	'm'	'magenta'
[0 1 1]	'c'	'cyan'
[1 0 0]	'r'	'red'
[0 1 0]	'g'	'green'
[0 0 1]	'b'	'blue'
[1 1 1]	'w'	'white'
[0 0 0]	'k'	'black'

Example: `'MarkerFaceColor',[0 .8 1]`

'MarkerSize' - Marker size

6 (default) | scalar

Marker size, specified as the comma-separated pair consisting of `'MarkerSize'` and a scalar in points.

Example: `'MarkerSize',3.75`

Output Arguments

h - Stemseries object handle

scalar or column vector

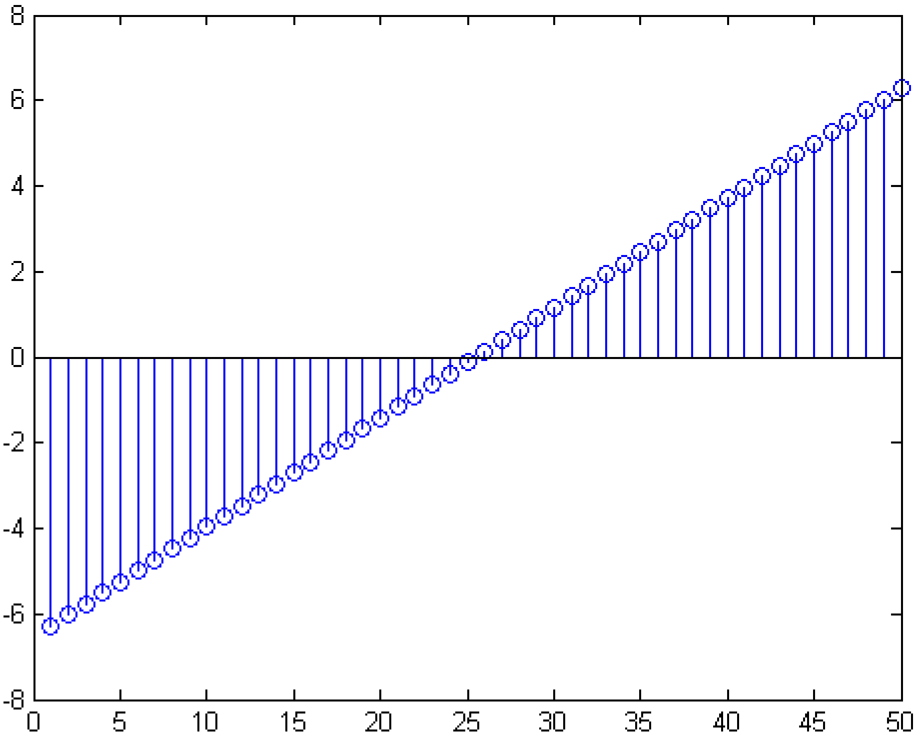
Stemseries object handle, specified as a scalar or column vector. This is a unique identifier, which you can use to query and modify the properties of a specific `stemseries`.

Examples

Plot Single Data Series

Plot 50 data values between -2π and 2π with a vector input.

```
figure
Y = linspace(-2*pi,2*pi,50);
stem(Y)
```



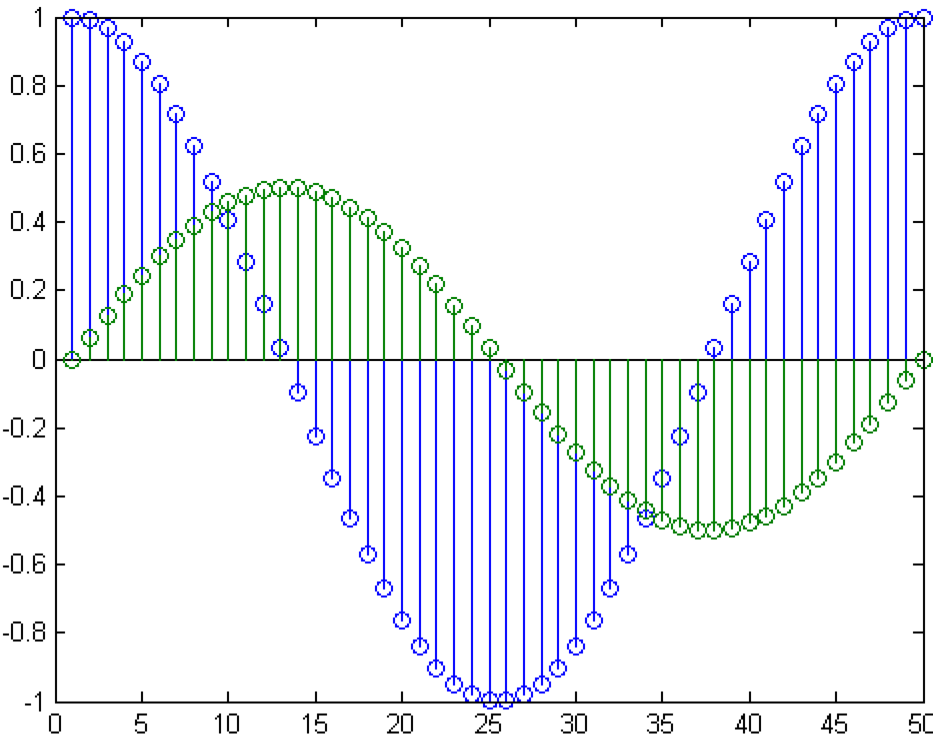
Data values are plotted as stems extending from the baseline and terminating at the data value. The length of `Y` automatically determines the position of each stem on the *x*-axis.

Plot Multiple Data Series

Plot two data series with a two-column matrix input.

```
figure
X = linspace(0,2*pi,50)';
Y = [cos(X), 0.5*sin(X)];
```

```
stem(Y)
```

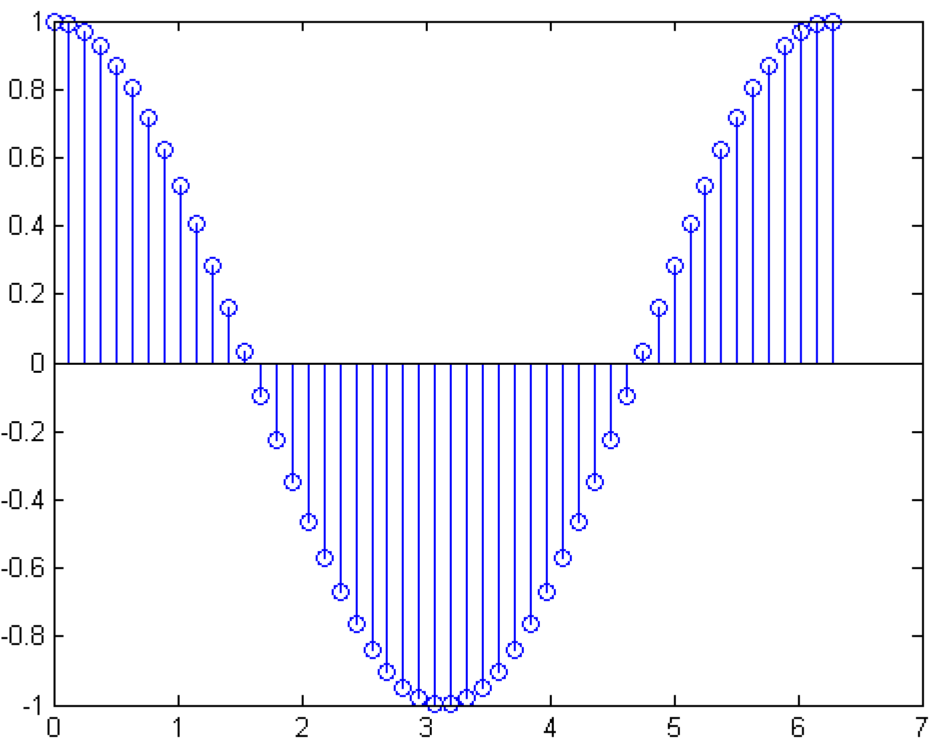


Each column of `Y` is plotted as a separate series, and entries in the same row of `Y` are plotted against the same *x* value. The number of rows in `Y` automatically generates the position of each stem on the *x*-axis.

Plot Single Data Series at Specified *x* values

Plot 50 data values of cosine evaluated between 0 and 2π and specify the set of *x* values for the stem plot.

```
figure
X = linspace(0,2*pi,50)';
Y = cos(X);
stem(X,Y)
```

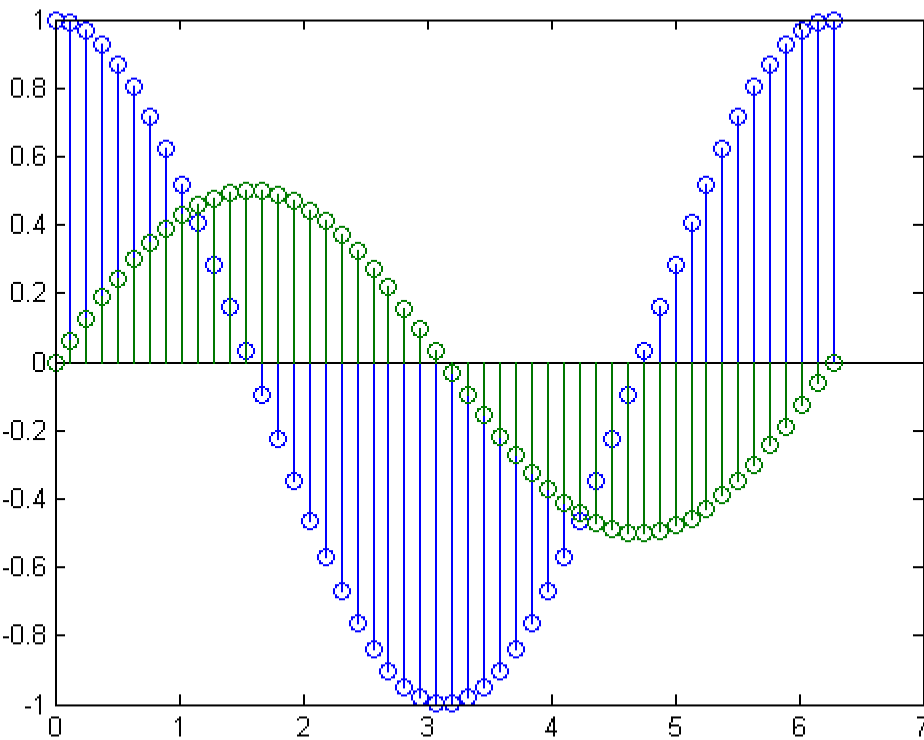


The first vector input determines the position of each stem on the *x*-axis.

Plot Multiple Data Series at Specified *x* values

Plot 50 data values of sine and cosine evaluated between 0 and 2π and specify the set of *x* values for the stem plot.

```
figure
X = linspace(0,2*pi,50)';
Y = [cos(X), 0.5*sin(X)];
stem(X,Y)
```

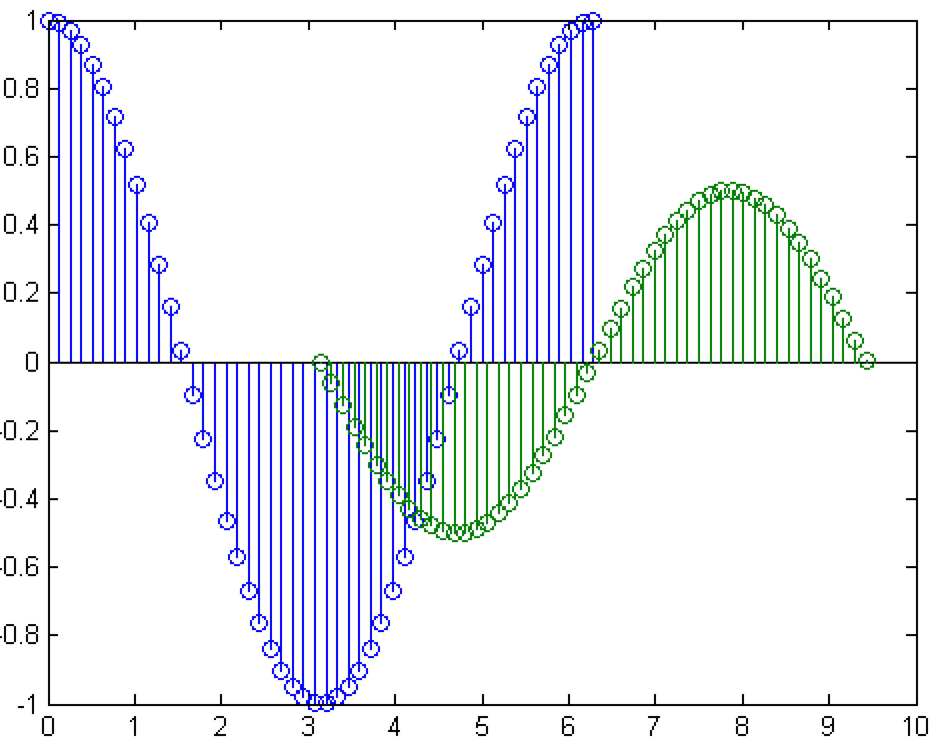


The vector input determines the *x*-axis positions for both data series.

Plot Multiple Data Series at Unique Sets of *x* values

Plot 50 data values of sine and cosine evaluated at different sets of *x* values. Specify the corresponding sets of *x* values for each series.

```
figure
x1 = linspace(0,2*pi,50)';
x2 = linspace(pi,3*pi,50)';
X = [x1, x2];
Y = [cos(x1), 0.5*sin(x2)];
stem(X,Y)
```

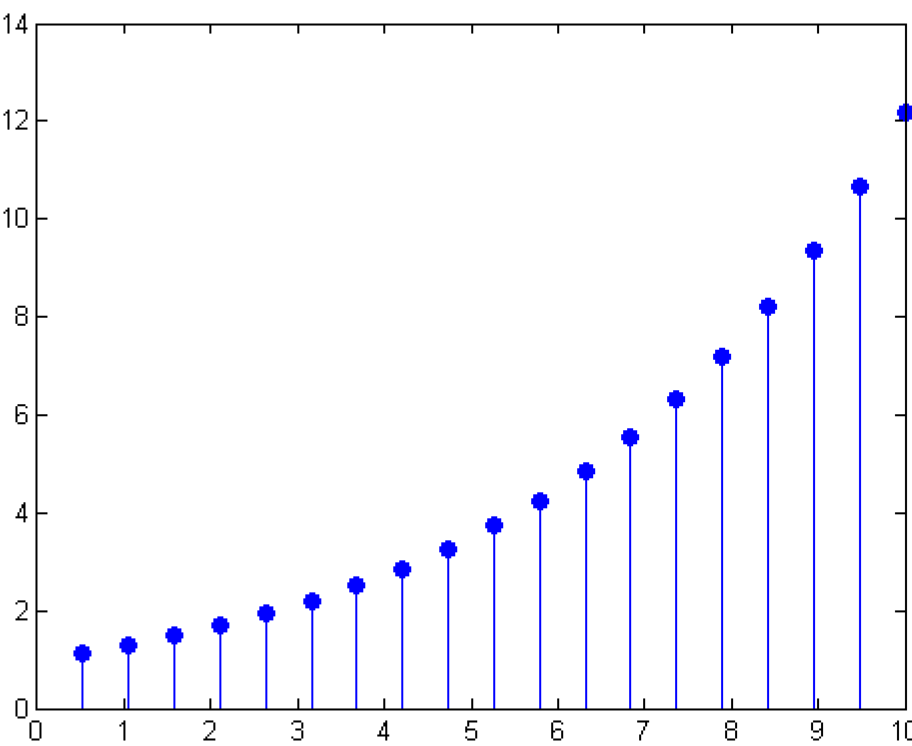


Each column of `X` is plotted against the corresponding column of `Y`.

Fill in Plot Markers

Create a stem plot and fill the circles that terminate each stem.

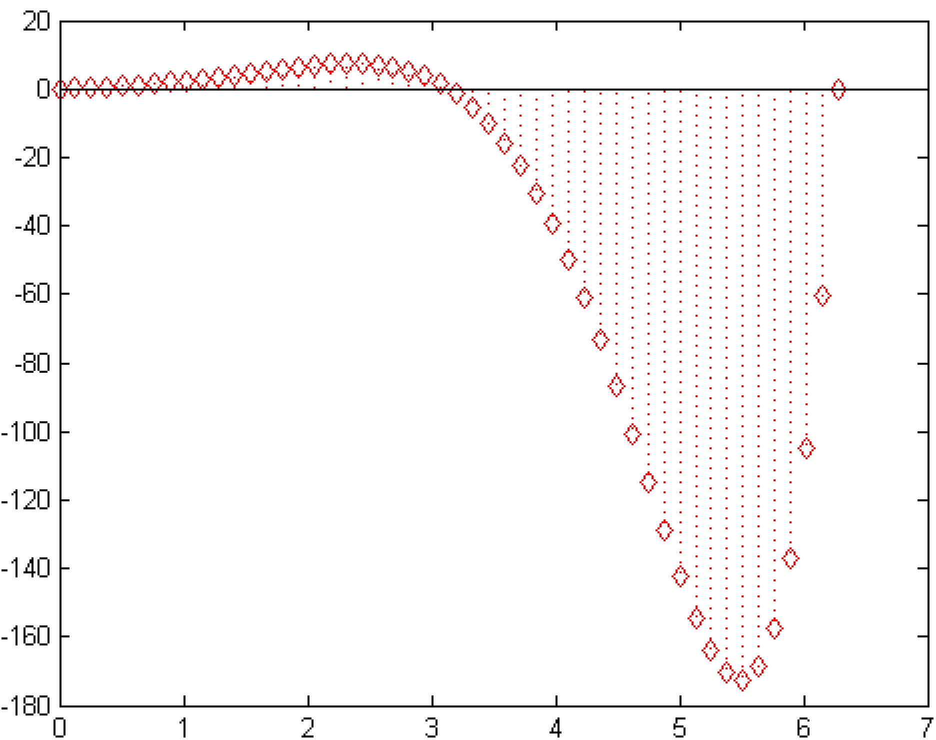
```
figure
X = linspace(0,10,20)';
Y = (exp(0.25*X));
stem(X,Y,'fill')
```



Specify Stem and Marker Options

Create a stem plot and set the line style to a dotted line, the marker symbols to diamonds, and the color to red using the `LineSpec` option.

```
figure
X = linspace(0,2*pi,50)';
Y = (exp(X).*sin(X));
stem(X,Y,':diamondr')
```

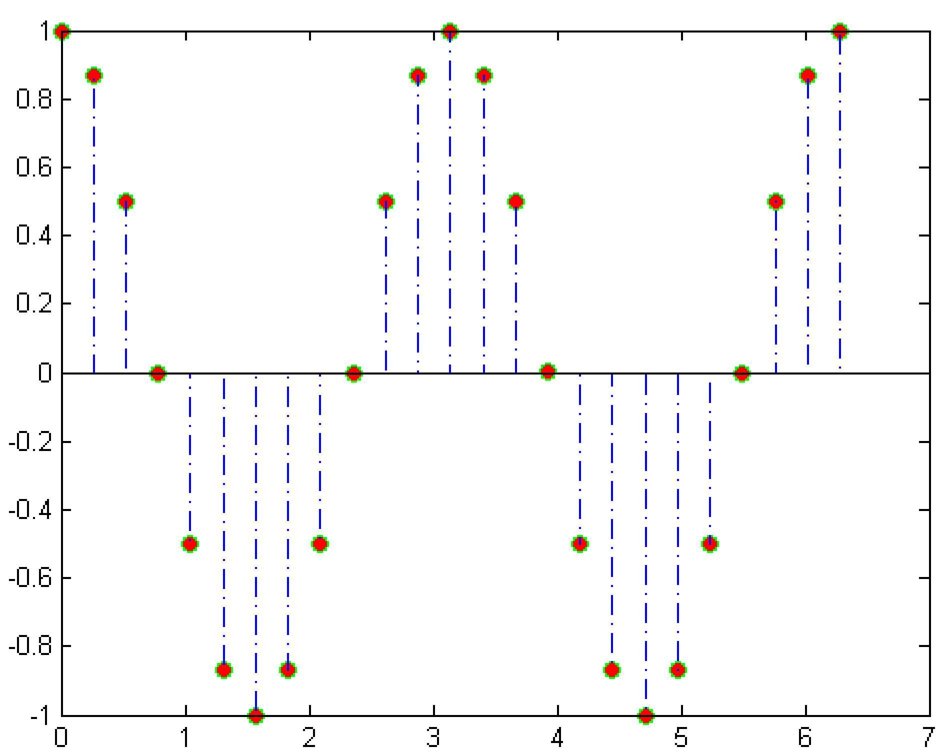


To color the inside of the diamonds, use the `'fill'` option.

Specify Additional Stem and Marker Options

Create a stem plot and set the line style to a dot-dashed line, the marker face color to red, and the marker edge color to green using `Name,Value` pair arguments.

```
figure
X = linspace(0,2*pi,25)';
Y = (cos(2*X));
stem(X,Y,'LineStyle','-.','MarkerFaceColor','red','MarkerEdgeColor','g
```



The stem remains the default color.

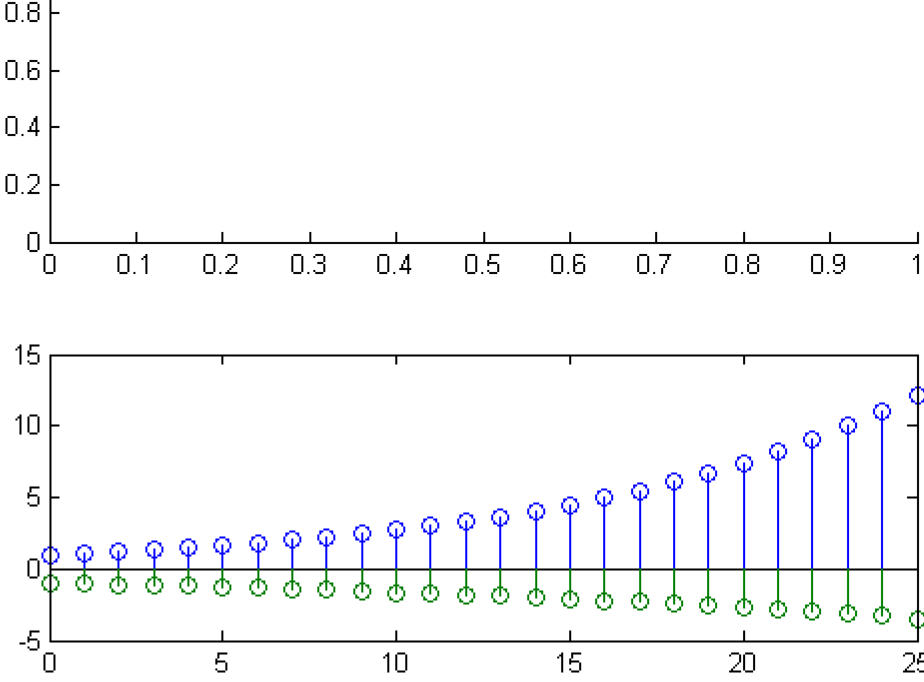
Specify Axes for Stem Plot

Create two subplots and return the handles to each axes, `s(1)` and `s(2)`.

```
figure
s(1) = subplot(2,1,1);
s(2) = subplot(2,1,2);
```

Create a stem plot in the lower subplot by referring to the axes handle, `s(2)`.

```
X = 0:25;
Y = [exp(0.1*X); -exp(.05*X)]';
stem(s(2),X,Y)
```



Return Stemseries Handle

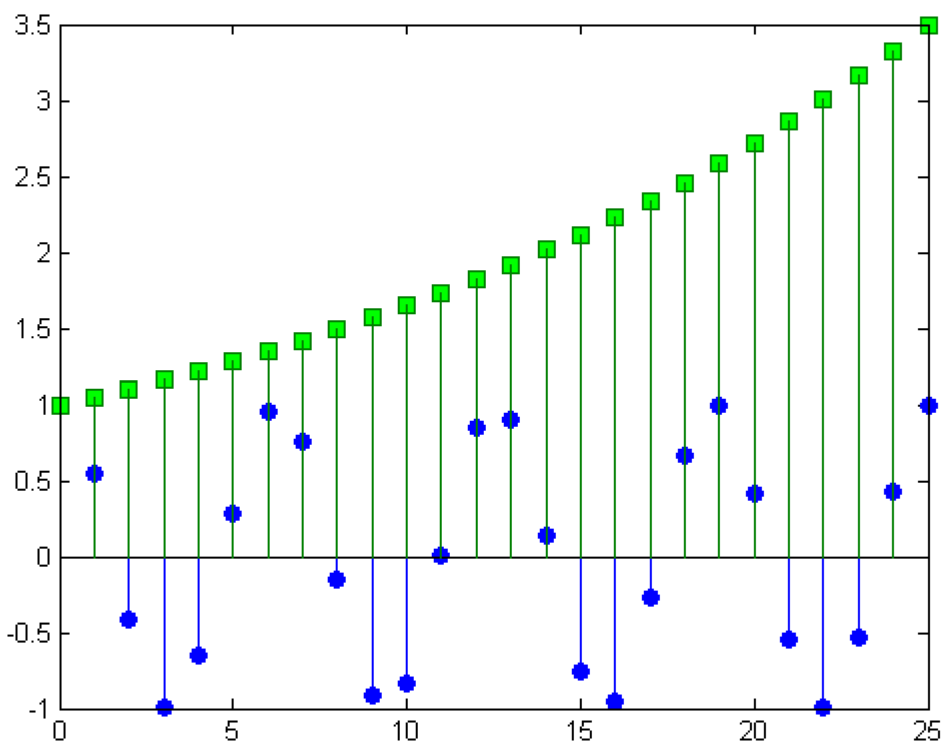
Create a stem plot with multiple series and return the handle to each series.

```
figure
X = 0:25;
Y = [cos(X); exp(0.05*X)]';
h = stem(X,Y);
```

The `stem` function creates two `stemseries` objects, one for each column of data. The output argument, `h`, returns both stemseries handles.

Change the markers of the first series to blue circles and the second series to green squares using the handles.

```
set(h(1),'MarkerFaceColor','blue')
set(h(2),'MarkerFaceColor','green','Marker','square')
```



Adjust Baseline Properties

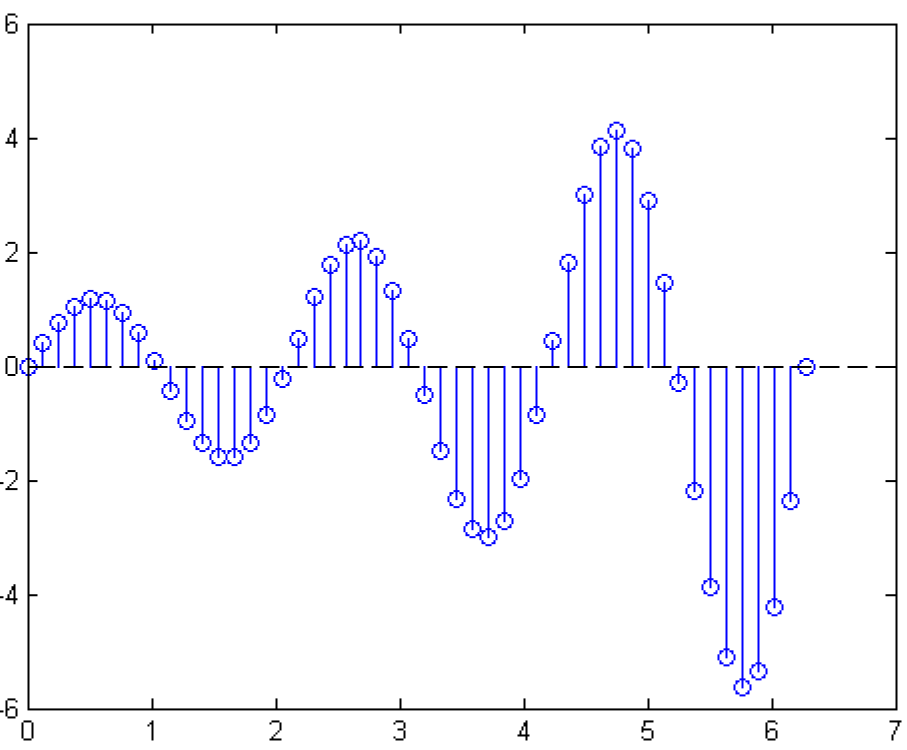
Create a stem plot and change properties of the baseline.

Get the baseline handle.

```
figure
X = linspace(0,2*pi,50)';
Y = (exp(0.3*X).*sin(3*X));
h = stem(X,Y);
hbase = get(h,'Baseline');
```

Adjust the color and style of the baseline by specifying a line property and value.

```
set(hbase,'LineStyle','--')
```



Make the baseline invisible by setting the visible property to `'off'`.

```
set(hbase,'Visible','off')
```

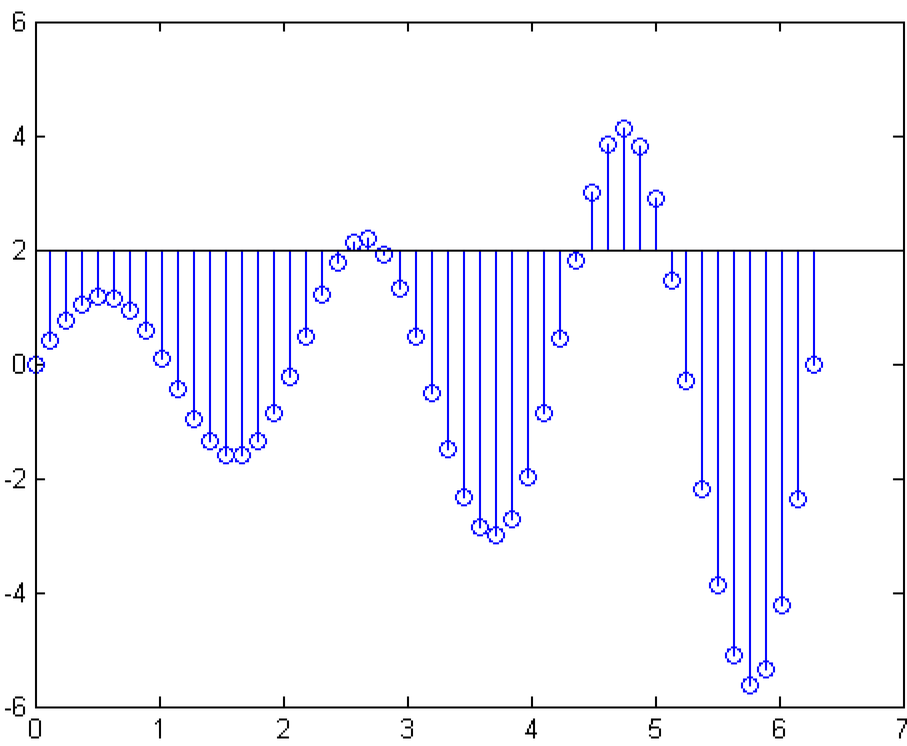
Delete the baseline using the delete command.

```
delete(hbase)
```

Change the Baseline Level

Create a stem plot and change the baseline level to 2.

```
figure
X = linspace(0,2*pi,50)';
Y = (exp(0.3*X).*sin(3*X));
h = stem(X,Y);
hbase = get(h,'Baseline');
set(hbase,'BaseValue',2)
```



See Also

bar | plot | stairs | LineSpecStemseries

stem3

Purpose

Plot 3-D discrete sequence data

Syntax

```
stem3(Z)
stem3(X,Y,Z)
stem3(___,'fill')
stem3(___,LineSpec)
stem3(___,Name,Value)

stem3(axes_handle,___)

h = stem3(___)
```

Description

`stem3(Z)` plots entries in `Z` as stems extending from the *xy*-plane and terminating with circles at the entry values. The stem locations in the *xy*-plane are automatically generated.

`stem3(X,Y,Z)` plots entries in `Z` as stems extending from the *xy*-plane where `X` and `Y` specify the stem locations in the *xy*-plane. The inputs `X`, `Y`, and `Z` must be vectors or matrices of the same size.

`stem3(___,'fill')` fills the circles. Use this option with any of the input argument combinations in the previous syntaxes.

`stem3(___,LineSpec)` specifies the line style, marker symbol, and color.

`stem3(___,Name,Value)` specifies stemseries properties using one or more `Name,Value` pair arguments.

`stem3(axes_handle,___)` plots into the axes specified by `axes_handle` instead of into the current axes (`gca`). The option, `axes_handle`, can precede any of the input argument combinations in the previous syntaxes.

h = stem3(___) returns the stemseries handle h.

Input Arguments

Z - Data sequence to display

vector or matrix

Data sequence to display, specified as a vector or matrix. stem3 plots each element in Z as a stem extending from the *xy*-plane and terminating at the data value.

- If Z is a row vector, stem3 plots all elements against the same *y* value at equally spaced *x* values.
- If Z is a column vector, stem3 plots all elements against the same *x* value at equally spaced *y* values.
- If Z is a matrix, stem3 plots each row of Z against the same *y* value at equally spaced *x* values.

Data Types

single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

X - Locations to plot values of Z

vector or matrix

Locations to plot values ofZ, specified as a vector or a matrix. Inputs X, Y and Z must be vectors or matrices of the same size.

Data Types

single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

Y - Locations to plot values of Z

vector or matrix

Locations to plot values of Z, specified as a vector or a matrix. Inputs X, Y and Z must be vectors or matrices of the same size.

Data Types

single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

LineSpec - Line style, marker symbol, and color

string

Line style, marker symbol, and color, specified as a string. For more information on line style, marker symbol, and color options see `LineSpec`.

Example: `':*r'`

Data Types
`char`

axes_handle - Axes handle

Axes handle, which is the reference to an axes object. Use the `gca` function to get the handle to the current axes, for example, `axes_handle = gca;`.

Name-Value Pair Arguments

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

Example: `'LineStyle',':','MarkerFaceColor','red'` plots the stem as a dotted line and sets the marker face color to red.

For more information on these properties see `stemseries`.

'LineStyle' - Line style

`'-'` (default) | `'--'` | `':'` | `'-.'` | `'none'`

Line style, specified as the comma-separated pair consisting of `'LineStyle'` and a line style specifier. This table lists supported line styles.

Specifier	Line Style
'- '	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-. '	Dash-dot line
'none'	No line

Example: 'LineStyle','-.'

'LineWidth' - Line width

0.5 (default) | scalar

Line width, specified as the comma-separated pair consisting of 'LineWidth' and a scalar. The scalar sets the width size in points of the stem and marker edge.

Example: 'LineWidth',0.75

'Color' - Color

[0 0 1] (blue) (default) | 3-element RGB vector | string

Color, specified as the comma-separated pair consisting of 'Color' and a 3-element RGB vector or a string containing the short or long name of the color. The RGB vector is a 3-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0 1].

This table lists the predefined colors and their RGB equivalents.

RGB Vector	Short Name	Long Name
[1 1 0]	'y'	'yellow'
[1 0 1]	'm'	'magenta'

RGB Vector	Short Name	Long Name
[0 1 1]	'c'	'cyan'
[1 0 0]	'r'	'red'
[0 1 0]	'g'	'green'
[0 0 1]	'b'	'blue'
[1 1 1]	'w'	'white'
[0 0 0]	'k'	'black'

Example: 'Color',[0 1 0]

Example: 'Color','green'

'Marker' - Marker symbol

'o' (default) | string

Marker symbol, specified as the comma-separated pair consisting of 'Marker' and a marker specifier. This table lists supported marker symbols.

Specifier	MarkerSymbol
'o'	Circle
'+'	Plus sign
'*'	Asterisk
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle

Specifier	MarkerSymbol
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No marker

Example: 'Marker','+'

Example: 'Marker','diamond'

'MarkerEdgeColor' - Marker edge color

[0 0 1] (blue) (default) | 'auto' | 'none' | three-element RGB vector | string

Marker edge color, specified as the comma-separated pair consisting of 'MarkerEdgeColor' and a color value. The color value can be one of the supported strings or an RGB vector, listed in the following tables.

Specifier	Result
'auto'	Uses same color as line color
'none'	Specifies no color, which makes unfilled markers invisible

For an RGB vector, use a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0 1]. This table lists the predefined string colors and their RGB equivalents.

RGB Vector	Short Name	Long Name
[1 1 0]	'y'	'yellow'
[1 0 1]	'm'	'magenta'

RGB Vector	Short Name	Long Name
[0 1 1]	'c'	'cyan'
[1 0 0]	'r'	'red'
[0 1 0]	'g'	'green'
[0 0 1]	'b'	'blue'
[1 1 1]	'w'	'white'
[0 0 0]	'k'	'black'

Example: `'MarkerEdgeColor',[1 .8 .1]`

'MarkerFaceColor' - Marker face color

`'none'` (default) | `'auto'` | three-element RGB vector | string

Marker face color, specified as the comma-separated pair consisting of `'MarkerFaceColor'` and a color value. `MarkerFaceColor` sets the fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles). The color value can be one of the supported strings or an RGB vector, listed in the following tables.

Specifier	Result
'auto'	Uses same color as marker edge color
'none'	Makes the interior of the marker transparent, allowing the background to show through (default)

For an RGB vector, use a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0 1]`. This table lists the predefined string colors and their RGB equivalents.

RGB Vector	Short Name	Long Name
[1 1 0]	'y'	'yellow'
[1 0 1]	'm'	'magenta'
[0 1 1]	'c'	'cyan'
[1 0 0]	'r'	'red'
[0 1 0]	'g'	'green'
[0 0 1]	'b'	'blue'
[1 1 1]	'w'	'white'
[0 0 0]	'k'	'black'

Example: `'MarkerFaceColor',[0 .8 1]`

'MarkerSize' - Marker size

6 (default) | scalar

Marker size, specified as the comma-separated pair consisting of `'MarkerSize'` and a scalar in points.

Example: `'Markersize',12`

Output Arguments

h - Stemseries object handle

scalar

Stemseries object handle, specified as a scalar. This is a unique identifier, which you can use to query and modify the properties of the `stemseries`.

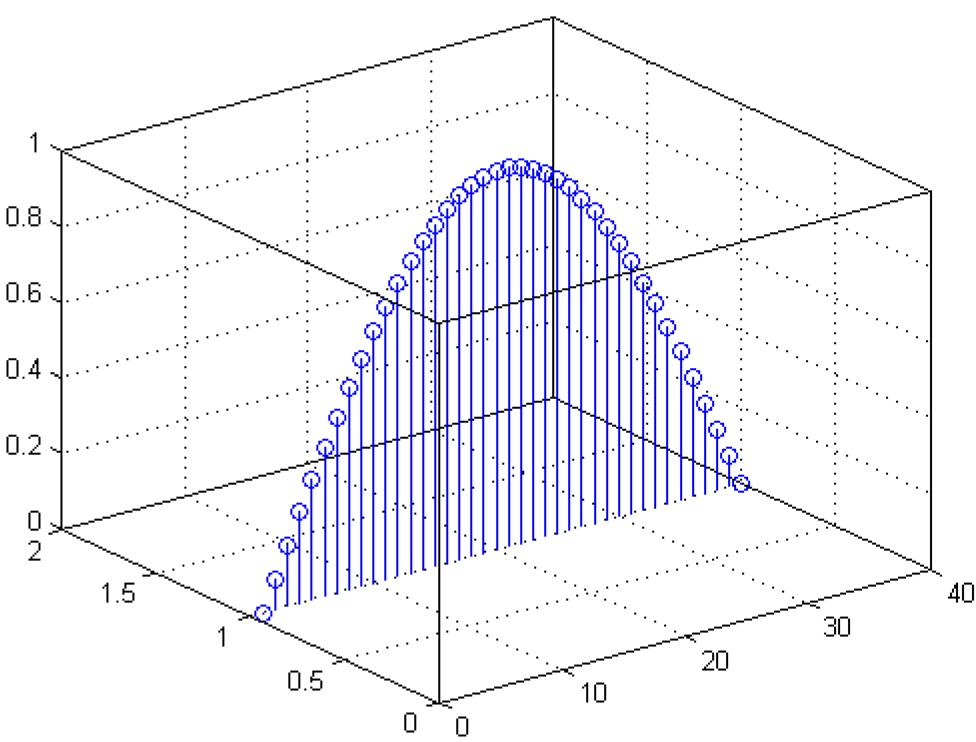
Examples

Row Vector Input

Create a 3-D stem plot of cosine values between -π/2 and π/2 with a row vector input.

```
figure
X = linspace(-pi/2,pi/2,40);
Z = cos(X);
```

```
stem3(Z)
```



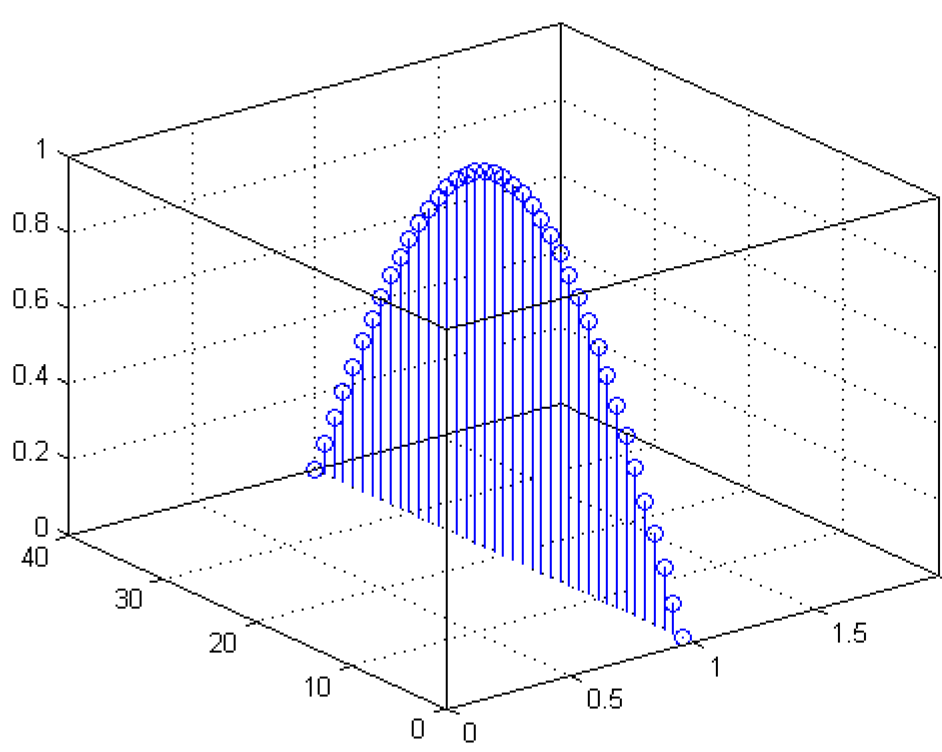
`stem3` plots elements of `Z` against the same *y* value at equally space *x* values.

Column Vector Input

Create a 3-D stem plot of cosine values between -π/2 and π/2 with a column vector input.

```
figure
```

```
X = linspace(-pi/2,pi/2,40)';
Z = cos(X);
stem3(Z)
```

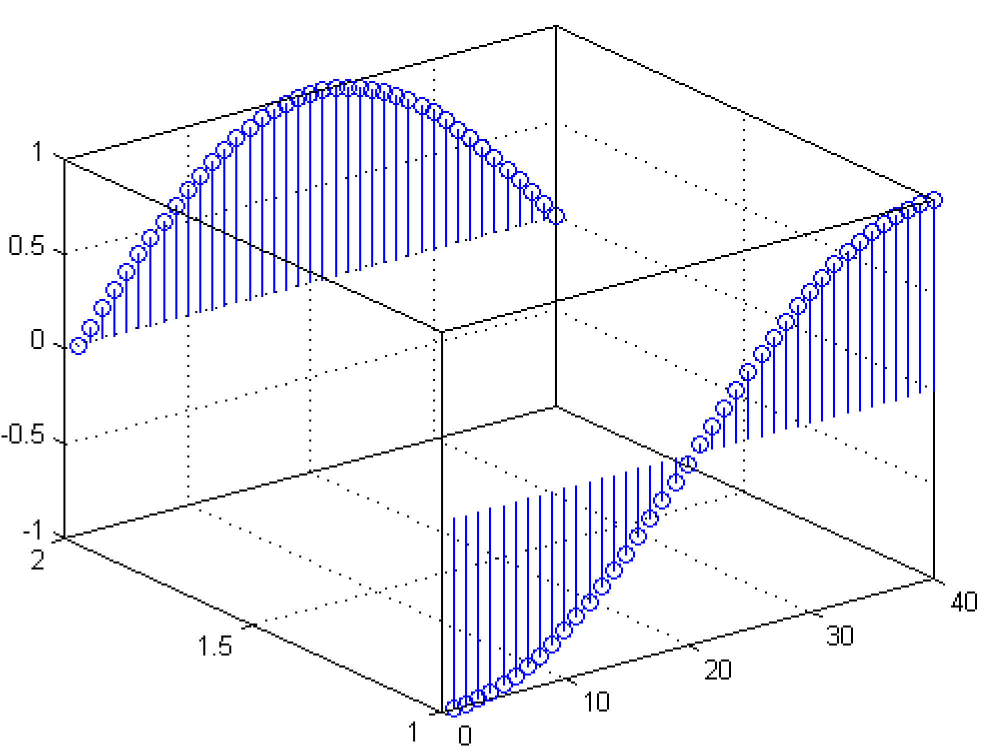


`stem3` plots elements of `Z` against the same *x* value at equally space *y* values.

Matrix Input

Create a 3-D stem plot of sine and cosine values between -π/2 and π/2 with a matrix input.

```
figure
X = linspace(-pi/2,pi/2,40);
Z = [sin(X); cos(X)];
stem3(Z)
```

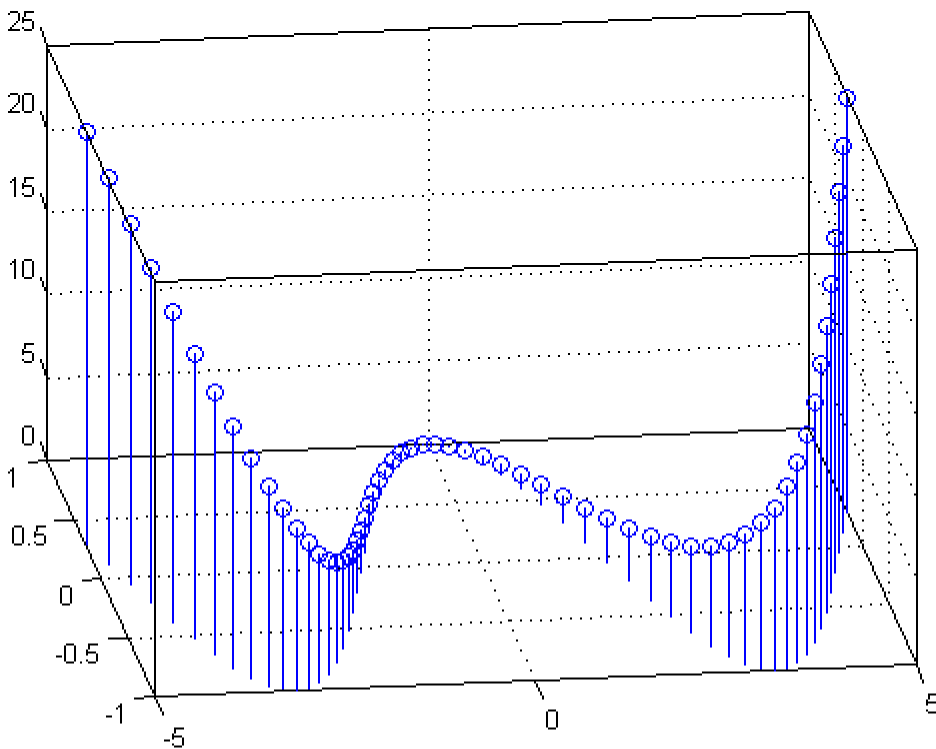


`stem3` plots each row of `Z` against the same *y* value at equally space *x* values.

Specify Stem Locations with Vector Inputs

Create a 3-D stem plot and specify the stem locations along a curve. Use `view` to adjust the angle of the axes in the figure.

```
figure
X = linspace(-5,5,60);
Y = cos(X);
Z = X.^2;
stem3(X,Y,Z)
view(-8,30)
```

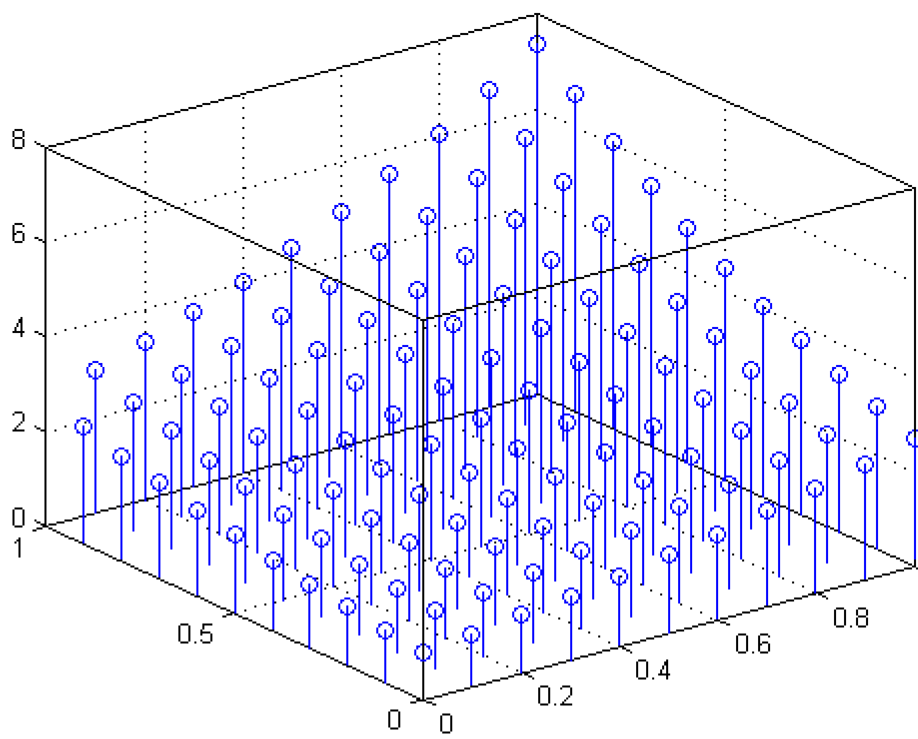


X and Y determine the stem locations. Z determines the marker heights.

Specify Stem Locations with Matrix Inputs

Create a 3-D stem plot with matrix data and specify the stem locations in the *xy*-plane.

```
figure
[X,Y] = meshgrid(0:.1:1);
Z = exp(X+Y);
stem3(X,Y,Z)
```

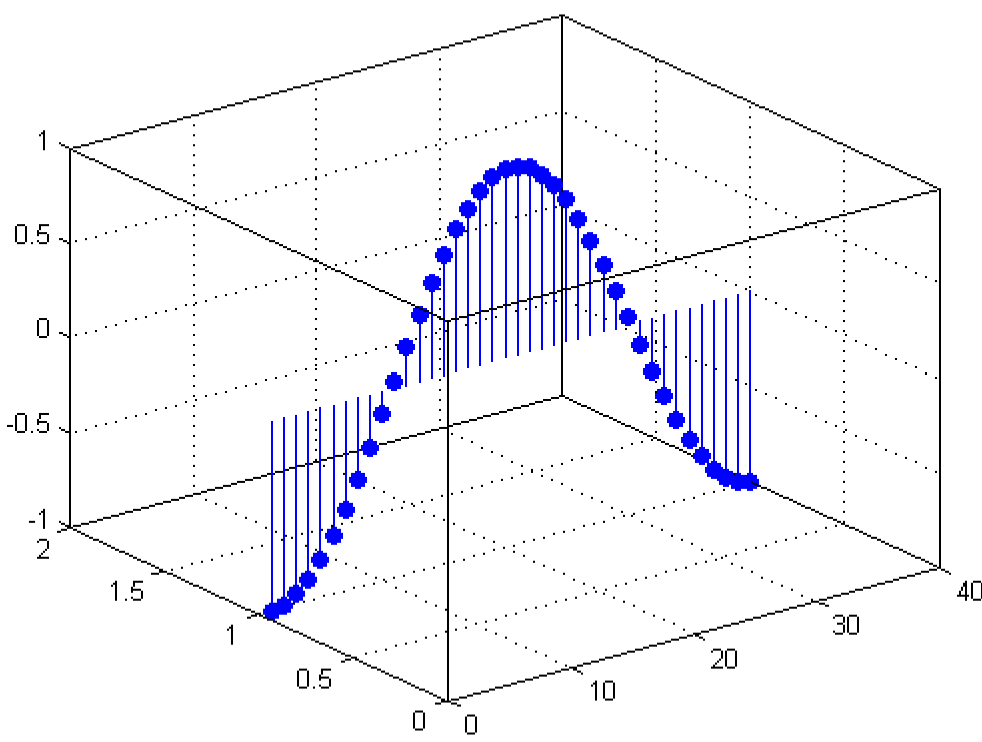


X and Y determine the stem locations. Z determines the marker heights.

Fill in the Circles

Create a 3-D stem plot of cosine values between -π and π. Use 'fill' to fill in the circles.

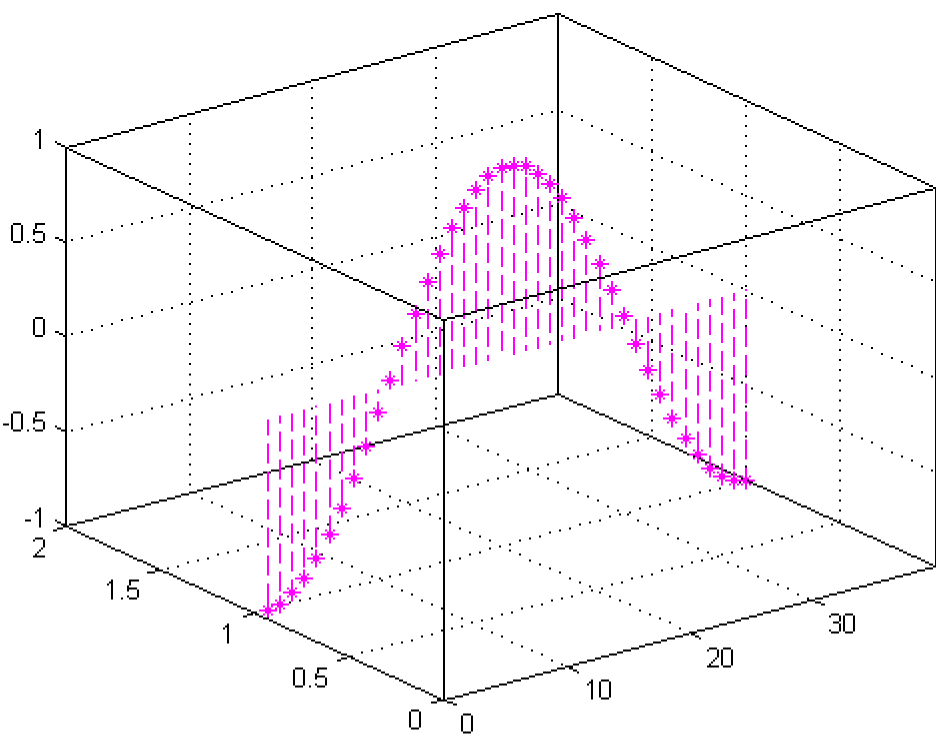
```
figure
X = linspace(-pi,pi,40);
Z = cos(X);
stem3(Z,'fill')
```



Line Style, Marker Symbol, and Color Options

Create a 3-D stem plot of cosine values between -π and π. Set the stem to a dashed line style, the marker symbols to stars, and the color to magenta using `LineSpec`.

```
figure
X = linspace(-pi,pi,40);
Z = cos(X);
stem3(Z,'--*m')
```



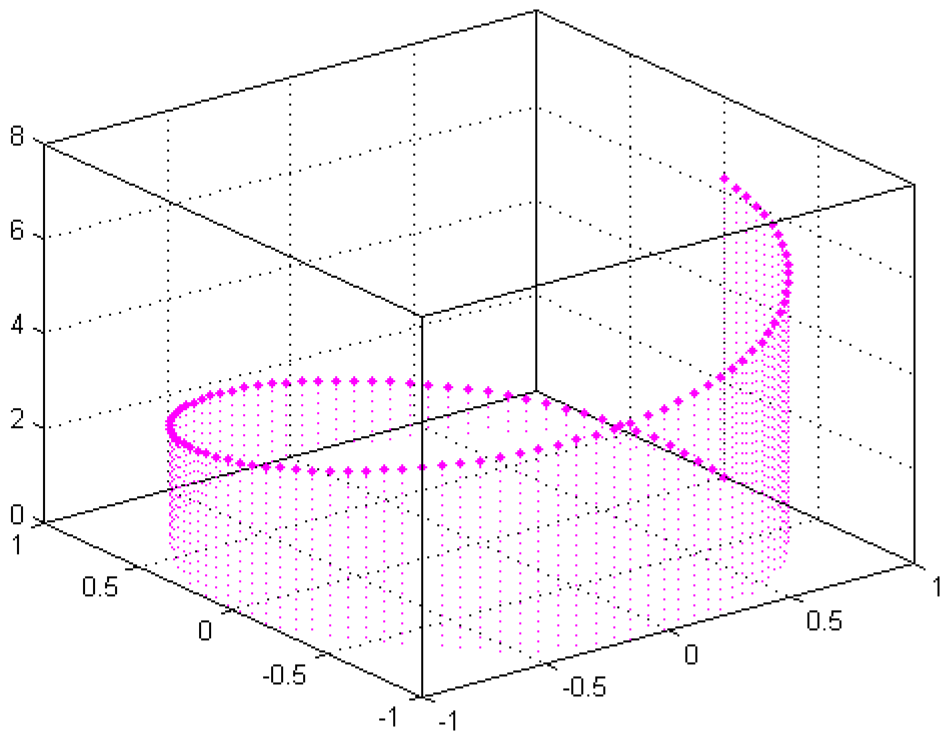
To specify only two of the three `LineSpec` options, omit the third option from the string. For example,`'*m'` sets the marker symbol and the color and leaves the default line style.

Line Style, Marker Symbol, and Color Options

Create a 3-D stem plot and specify the stem locations along a circle. Set the stem to a dashed line style, the marker symbols to stars, and the color to magenta using `LineSpec`.

```
figure
```

```
theta = linspace(0,2*pi);
X = cos(theta);
Y = sin(theta);
Z = theta;
stem3(X,Y,Z,':.m')
```

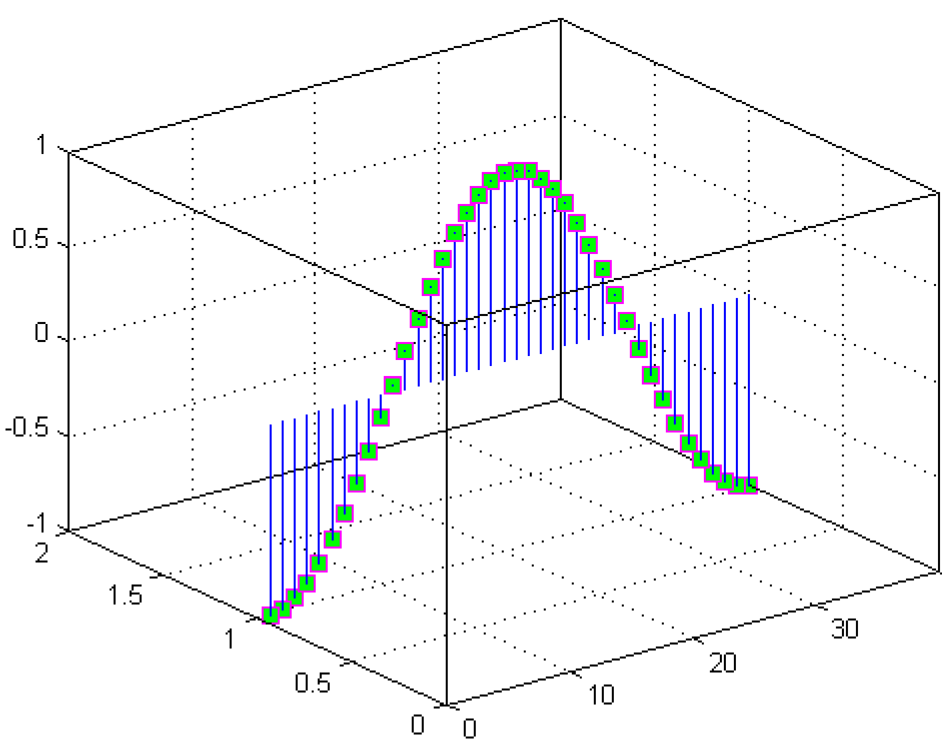


X and Y determine the stem locations. Z determines the marker heights.

Additional Style Options

Create a 3-D stem plot of cosine values between -π and π. Set the marker symbols to squares with green faces and magenta edges.

```
figure
X = linspace(-pi,pi,40);
Z = cos(X);
stem3(Z,'Marker','s','MarkerEdgeColor','m','MarkerFaceColor','g')
```



Axes Handles

Specify an axes for the 3-D stem plot.

Create two subplots and return the handles to each axes, `s(1)` and `s(2)`.

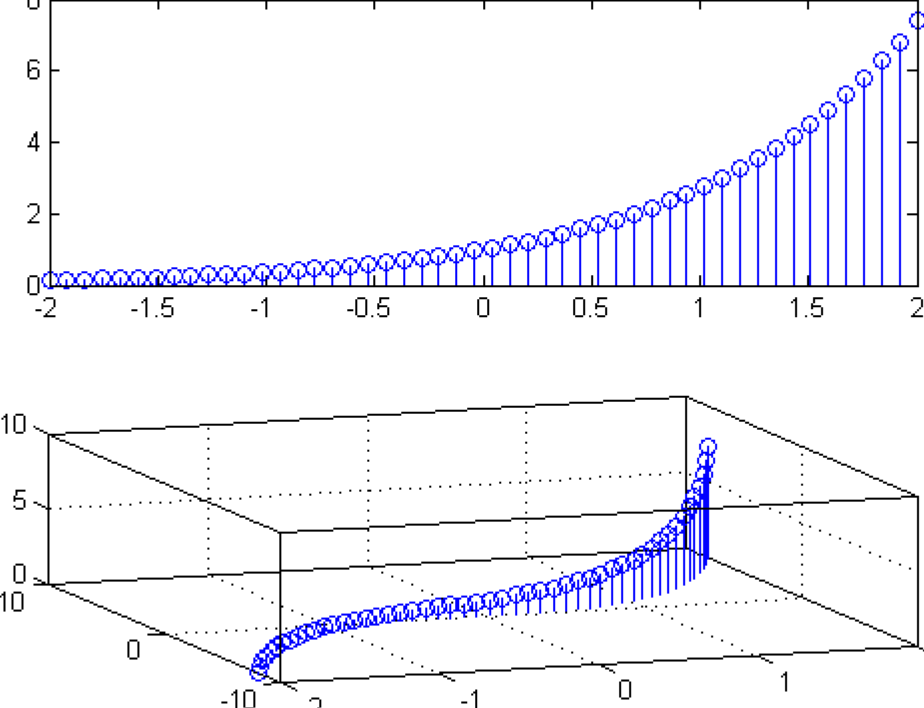
```
figure
s(1) = subplot(2,1,1);
s(2) = subplot(2,1,2);
```

Define vectors `X`, `Y` and `Z`.

```
X = linspace(-2,2,50);
Y = X.^3;
Z = exp(X);
```

Plot a 3-D stem plot in the lower subplot using the axes handle `s(2)`. Adjust the angle of the axes using `view`. For comparison, plot a 2-D stem plot in the upper subplot using `X` and `Z`.

```
stem(s(1),X,Z)
stem3(s(2),X,Y,Z)
view(-20,35)
```



Stemseries Handle

Create a 3-D stem plot and use the stemseries handle to adjust properties of the plot.

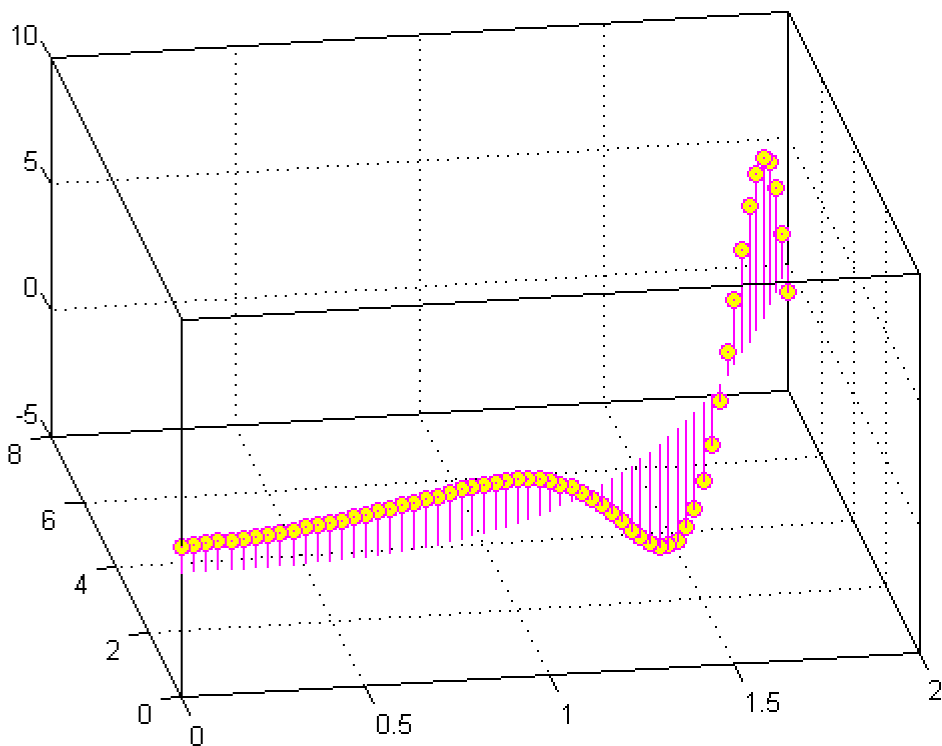
Get the handle for the stemseries.

```
figure
X = linspace(0,2);
Y = X.^3;
Z = exp(X).*cos(Y);
```

```
h = stem3(X,Y,Z,'fill');
```

Set the color to magenta and the marker face color to yellow. Use `view` to adjust the angle of the axes in the figure.

```
set(h,'Color','m','MarkerFaceColor','y');
view(-10,35)
```



See Also

`bar` | `plot` | `stairs` | `stemStemseries Properties`

Related Examples

- "Discrete Data Graphs"

Stemseries Properties

Purpose

Define stemseries properties

Modifying Properties

You can set and query graphics object properties using the `set` and `get` commands or with the property editor (`propertyeditor`).

Note that you cannot define default properties for stemseries objects.

See Plot Objects for information on stemseries objects.

Stemseries Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`Annotation`

`hg.Annotation` object (read-only)

Control the display of stemseries objects in legends. Specifies whether this stemseries object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the stemseries object is displayed in a figure legend.

IconDisplayStyle Value	Purpose
`on`	Include the stemseries object in a legend as one entry, but not its children objects
`off`	Do not include the stemseries or its children in a legend (default)
`children`	Include only the children of the stemseries as separate entries in the legend

Setting the IconDisplayStyle Property

These commands set the `IconDisplayStyle` of a graphics object with handle `hobj` to `children`, which causes each child object to have an entry in the legend:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','children')
```

Using the IconDisplayStyle Property

See "Controlling Legends" for more information and examples.

`BaseLine`
handle

Handle of the baseline object. Handle of the line object used as the baseline. Set the properties of this line using its handle.

The following example makes the stem plot baseline a dashed, red line:

```
% Create stem plot
stem_handle = stem(randn(10,1));
% Obtain handle of baseline from stemseries object
baseline_handle = get(stem_handle,'BaseLine');
% Set line properties
set(baseline_handle,'LineStyle','--','Color','red')
```

`BaseValue`
y-axis value

Y-axis value where baseline is drawn. Specify the value along the *y*-axis at which the MATLAB software draws the baseline.

`BeingDeleted`
`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`

string | function handle

Button press callback function. Executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object. See the `HitTestArea` property for information about selecting objects of this type.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be:

- A string that is a valid MATLAB expression
- The name of a MATLAB file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`Children`

array of graphics object handles

Children of the stemseries object. An array containing the handles of all line objects parented to the stemseries object (whether visible or not).

If a child object's `HandleVisibility` property is `callback` or `off`, its handle does not show up in this object's `Children` property. If you want the handle in the `Children` property, set the root `ShowHiddenHandles` property to `on`. For example:

```
set(0,'ShowHiddenHandles','on')
```

`Clipping`

`{on} | off`

Clipping mode. MATLAB clips graphs to the axes plot box by default. If you set `Clipping` to `off`, portions of graphs appear outside the axes plot box. This occurs if you create a plot object, set `hold` to `on`, freeze axis scaling (`axis manual`), and then create a larger plot object.

`Color`

`ColorSpec`

Color of stem lines. A three-element RGB vector or one of the MATLAB predefined names, specifying the line color. See the `ColorSpec` reference page for more information on specifying color.

For example, the following statement creates a stem plot with red lines.

```
h = stem(randn(10,1),'Color','r');
```

`CreateFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback function executed during object creation. Defines a callback that executes when MATLAB creates an object. The default is an empty array.

You must specify the callback during the creation of the object. For example:

```
graphicfcn(y,'CreateFcn',@CallbackFcn)
```

where `@CallbackFcn` is a function handle that references the callback function and `graphicfcn` is the plotting function which creates this object.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DeleteFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback executed during object deletion. Executes when this object is deleted (for example, this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values. The default is an empty array.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DisplayName`

string

String used by legend. The `legend` function uses the `DisplayName` property to label the stemseries object in the legend. The default is an empty string.

- If you specify string arguments with the `legend` function, MATLAB set `DisplayName` to the corresponding string and uses that string for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, MATLAB set `DisplayName` to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add a legend programmatically that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more information and examples.

`EraseMode`
`{normal} | none | xor | background`

Erase mode. Controls the technique MATLAB uses to draw and erase objects. Use alternative erase modes for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the object when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it because MATLAB stores no information about its former location.

- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object's color depends on the color of whatever is beneath it on the display.
- `background` — Erase the object by redrawing it in the axes background `Color`, or the figure background `Color` if the axes `Color` property is `none`. This damages objects that are behind the erased object, but properly colors the erased object.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors (for example, performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible.

- callback — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- off — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes get, findobj, gca, gcf, gco, newplot, cla, clf, and close.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using callback or off, the object's handle does not appear in its parent's Children property, figures do not appear in the root's CurrentFigure property, objects do not appear in the root's CallbackObject property or in the figure's CurrentObject property, and axes do not appear in their parent's CurrentAxes property.

Overriding Handle Visibility

You can set the root ShowHiddenHandles property to on to make all handles visible regardless of their HandleVisibility settings. This does not affect the values of the HandleVisibility properties. See also findall.

Handle Validity

Hidden handles are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`
`{on} | off`

Selectable by mouse click. Determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the line objects that compose the stem plot. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`HitTestArea`
`on | {off}`

Select the object by clicking stem lines or area of extent. Select plot objects by:

- Clicking stem lines (default).
- Clicking anywhere in the extent of the plot.

When `HitTestArea` is `off`, you must click the stem lines to select the object. When `HitTestArea` is `on`, you can select this object by clicking anywhere within the extent of the plot (that is, anywhere within a rectangle that encloses all the stem lines).

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the `Interruptible` property affects only the callbacks for the`ButtonDownFcn` property. A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both the callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see "Control Callback Execution and Interruption".

`LineStyle`
`{-} | -- | : | -. | none`

Line style of stemseries object.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line
'none'	No line

Use `LineStyle` `none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`
size in points

Width of linear objects and edges of filled areas. Specify in points. 1 point = $^1/_{72}$ inch. The default is `0.5` points.

`Marker`
character (see table)

Marker symbol. Specifies marks that display at data points. You can set values for the `Marker` property independently from the `LineStyle` property. For a list of supported marker symbols, see the following table.

Marker Specifiers Table

Specifier	Marker Type
'+'	Plus sign
'o'	Circle
'*'	Asterisk
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'' '	Six-pointed star (hexagram)
'none'	No marker (default)

`MarkerEdgeColor`
`ColorSpec | none | {auto}`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `ColorSpec` — User-defined color.
- `none` — Specifies no color, which makes nonfilled markers invisible.
- `auto` — Uses same color as the `Color` property.

`MarkerFaceColor`
`ColorSpec | {none} | auto`

Fill color for closed-shape markers. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `ColorSpec` — User-defined color.
- `none` — Makes the interior of the marker transparent, allowing the background to show through.
- `auto` — Sets the fill color to the axes `Color` property. If the axes `Color` property is `none`, sets the fill color to the figure `Color`.

`MarkerSize`
scalar

Marker size. Size of the marker in points. The default value is `6`.

Note MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`Parent`
handle of parent axes, hggroup, or hgtransform

Parent of object. Handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
: `on | {off}`

Object selection state. When you set this property to `on`, MATLAB displays selection handles at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
: `{on} | off`

Object highlighted when selected.

- `on` — MATLAB indicates the selected state by drawing four edge handles and four corner handles.
- `off` — MATLAB does not draw the handles except when in plot edit mode and objects are selected manually.

`Tag`
: string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

For example, create a stemseries object and set the `Tag` property:

```
Y = linspace(-2*pi,2*pi,10);
t = stem(Y,'Tag','stem1')
```

To access the stemseries object, use `findobj` to find the object's handle. The following statement changes the `MarkerFaceColor` property of the object whose `Tag` is `stem1`.

```
set(findobj('Tag','stem1'),'MarkerFaceColor','red')
```

`Type`

string (read-only)

Type of graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For stemseries objects, `Type` is '`hggroup`'. The following statement finds all hggroup objects in the current axes object.

```
t = findobj(gca,'Type','hggroup');
```

`UIContextMenu`

handle of uicontextmenu object

Associate context menu with object. Handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object. The default value is an empty array.

`UserData`

array

User-specified data. Data you want to associate with this object (including cell arrays and structures). The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`
`{on} | off`

Visibility of object and its children.

- `on` — Object and all children of the object are visible unless the child object's `Visible` property is `off`.
- `off` — Object not displayed. However, the object still exists and you can set and query its properties.

`XData`
array

X-axis location of stems. The stem function draws an individual stem at each *x*-axis location in the `XData` array. `XData` can be either a matrix equal in size to `YData` or a vector equal in length to the number of rows in `YData`. That is, `length(XData) == size(YData,1)`. `XData` does not need to be monotonically increasing.

If you do not specify `XData` (which is the input argument `X`), the stem function uses the indices of `YData` to create the stem plot. See the `XDataMode` property for related information.

`XDataMode`
`{auto} | manual`

Use automatic or user-specified x-axis values. If you specify `XData` (by setting the `XData` property or specifying the `X` input argument), MATLAB sets this property to `manual` and uses the specified values to label the *x*-axis.

If you set `XDataMode` to `auto` after specifying `XData`, MATLAB resets the *x*-axis ticks to `1:size(YData,1)` or to the column indices of the `ZData`, overwriting any previous values for `XData`.

`XDataSource`
MATLAB variable, as a string

Link `XData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `XData`. The default value is an empty array.

```
set(h,'XDataSource','xdatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `XDataSource` does not change the object's `XData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`YData`

scalar | vector | matrix

Stem plot data. `YData` contains the data plotted as stems. Each value in `YData` is represented by a marker in the stem plot. If `YData` is a matrix, MATLAB creates a series of stems for each column in the matrix.

The input argument `Y` in the `stem` function calling syntax assigns values to `YData`.

`YDataSource`

MATLAB variable, as a string

Link `YData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `YData`. The default value is an empty array.

```
set(h,'YDataSource','Ydatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `YDataSource` does not change the object's `YData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`ZData`
vector

Z-coordinates. A data defining the stems for 3-D stem graphs. `XData` and `YData` (if specified) must be the same size.

`ZDataSource`
MATLAB variable, as a string

Link `ZData` *to MATLAB variable*. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `ZData`. The default value is an empty array.

```
set(h,'ZDataSource','zdatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `ZDataSource` does not change the object's `ZData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable

be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

---

stop

Purpose

Stop timer(s)

Syntax

```
stop(obj)
```

Description

`stop(obj)` stops the timer, represented by the timer object, `obj`. If obj is an array of timer objects, the `stop` function stops them all. Use the `timer` function to create a timer object.

The `stop` function sets the `Running` property of the timer object, `obj`, to `'off'`, halts further `TimerFcn` callbacks, and executes the `StopFcn` callback.

See Also

`timer` | `start`

Purpose

Stop asynchronous read and write operations

Syntax

```
stopasync(obj)
```

Description

`stopasync(obj)` stops any asynchronous read or write operation that is in progress for the serial port object, `obj`.

Tips

You can write data asynchronously using the `fprintf` or `fwrite` function. You can read data asynchronously using the `readasync` function, or by configuring the `ReadAsyncMode` property to `continuous`. In-progress asynchronous operations are indicated by the `TransferStatus` property.

If `obj` is an array of serial port objects and one of the objects cannot be stopped, the remaining objects in the array are stopped and a warning is returned. After an object stops:

- Its `TransferStatus` property is configured to `idle`.
- Its `ReadAsyncMode` property is configured to `manual`.
- The data in its output buffer is flushed.

Data in the input buffer is not flushed. You can return this data to the MATLAB workspace using any of the synchronous read functions. If you execute the `readasync` function, or configure the `ReadAsyncMode` property to `continuous`, then the new data is appended to the existing data in the input buffer.

See Also

`fprintf` | `fwrite` | `readasync` | `ReadAsyncMode` | `TransferStatus`

str2double

Purpose

Convert string to double-precision value

Syntax

```
X = str2double('str')
X = str2double(C)
```

Description

`X = str2double('str')` converts the string `str`, which should be an ASCII character representation of a real or complex scalar value, to the MATLAB double-precision representation. The string can contain digits, a comma (thousands separator), a decimal point, a leading `+` or `-` sign, an `e` preceding a power of 10 scale factor, and an `i` for a complex unit.

If `str` does not represent a valid scalar value, `str2double` returns `NaN`.

`X = str2double(C)` converts the strings in the cell array of strings `C` to double precision. The matrix `X` returned will be the same size as `C`.

Examples

Here are some valid `str2double` conversions.

```
str2double('123.45e7')
str2double('123 + 45i')
str2double('3.14159')
str2double('2.7i - 3.14')
str2double({'2.71' '3.1415'})
str2double('1,200.34')
```

See Also

`char` | `hex2num` | `num2str` | `str2num` | `cast`

Purpose

Construct function handle from function name string

Syntax

```
str2func('str')
```

Description

`str2func('str')` constructs a function handle `fhandle` for the function named in the string `'str'`. The contents of `str` can be the name of a file that defines a MATLAB function, or the name of an anonymous function.

You can create a function handle `fh` using any of the following four methods:

- Create a handle to a named function:

```
fh = @functionName;
fh = str2func(functionName);
```

- Create a handle to an anonymous function:

```
fh = @(x)functionDef(x);
fh = str2func('@(x)functionDef(x)');
```

You can create an array of function handles from strings by creating the handles individually with `str2func`, and then storing these handles in a cell array.

Tips

Nested functions are not accessible to `str2func`. To construct a function handle for a nested function, you must use the function handle constructor, `@`.

Any variables and their values originally stored in a function handle when it was created are lost if you convert the function handle to a string and back again using the `func2str` and `str2func` functions.

Examples

Example 1

To convert the string, `'sin'`, into a handle for that function, type

```
fh = str2func('sin')
```

```
fh =
    @sin
```

Example 2

If you pass a function name string in a variable, the function that receives the variable can convert the function name to a function handle using `str2func`. The example below passes the variable, `funcname`, to function `makeHandle`, which then creates a function handle. Here is the function:

```
function fh = makeHandle(funcname)
fh = str2func(funcname);
```

This is the code that calls `makeHandle` to construct the function handle:

```
makeHandle('sin')
ans =
    @sin
```

Example 3

To call `str2func` on a cell array of strings, use the `cellfun` function. This returns a cell array of function handles:

```
fh_array = cellfun(@str2func, {'sin' 'cos' 'tan'}, ...
                   'UniformOutput', false);

fh_array{2}(5)
ans =
   0.2837
```

Example 4

In the following example, the `myminbnd` function expects to receive either a function handle or string in the first argument. If you pass a string, `myminbnd` constructs a function handle from it using `str2func`, and then uses that handle in a call to `fminbnd`:

```
function myminbnd(fhandle, lower, upper)
if ischar(fhandle)
```

```
   disp 'converting function string to function handle ...'
   fhandle = str2func(fhandle);
end
fminbnd(fhandle, lower, upper)
```

Whether you call `myminbnd` with a function handle or function name string, the function can handle the argument appropriately:

```
myminbnd('humps', 0.3, 1)
converting function string to function handle ...
ans =
    0.6370
```

Example 5

The `dirByType` function shown here creates an anonymous function called `dirCheck`. What the anonymous function does depends upon the value of the `dirType` argument passed in to the primary function. The example demonstrates one possible use of `str2func` with anonymous functions:

```
function dirByType(dirType)
switch(dirType)
   case 'class', leadchar = '@';
   case 'package', leadchar = '+';
   otherwise disp('ERROR: Unrecognized type'), return;
end

dirfile = @(fs)isdir(fs.name);
dirCheckStr = ['@(fs)strcmp(fs.name(1,1),''', leadchar, ''')'];
dirCheckFun = str2func(dirCheckStr);
s = dir;   filecount = length(s);

for k=1:filecount
   fstruct = s(k);
   if dirfile(fstruct) && dirCheckFun(fstruct)
       fprintf('%s folder: %s\n', dirType, fstruct.name)
   end
end
```

Generate a list of class and package folders:

```
dirByType('class')
class folder: @Point
class folder: @asset
class folder: @bond

dirByType('package')
package folder: +containers
package folder: +event
package folder: +mypkg
```

See Also

`function_handle` | `func2str` | `functions`

Purpose

Form blank-padded character matrix from strings

Note `str2mat` is not recommended. Use `char` instead.

Syntax

```
S = str2mat(T1, T2, T3, ...)
```

Description

`S = str2mat(T1, T2, T3, ...)` forms the matrix `S` containing the text strings `T1, T2, T3, ...` as rows. The function automatically pads each string with blanks in order to form a valid matrix. Each text parameter, `Ti`, can itself be a string matrix. This allows the creation of arbitrarily large string matrices. Empty strings are significant.

Tips

`str2mat` differs from `strvcat` in that empty strings produce blank rows in the output. In `strvcat`, empty strings are ignored.

Examples

```
x = str2mat('36842', '39751', '38453', '90307');

whos x
  Name      Size          Bytes  Class

  x         4x5              40  char array

x(2,3)

ans =

    7
```

See Also

`char`

str2num

Purpose

Convert string to number

Syntax

```
x = str2num('str')
[x, status] = str2num('str')
```

Description

Note `str2num` uses the `eval` function to convert the input argument. Side effects can occur if the string contains calls to functions. Using `str2double` can avoid some of these side effects.

`x = str2num('str')` converts the string `str`, which is an ASCII character representation of a numeric value, to numeric representation. `str2num` also converts string matrices to numeric matrices. If the input string does not represent a valid number or matrix, `str2num(str)` returns the empty matrix in `x`.

The input string can contain one or more numbers separated by spaces, commas, or semicolons, such as '`5`', '`10,11,12`', or '`5,10;15,20`'. In addition to numerical values and delimiters, the input string can also include a decimal point, leading + or - signs, the letter `e` or `d` preceding a power of 10 scale factor, or the letter `i` or `j` indicating a complex or imaginary number.

The following table shows several examples of valid inputs to `str2num`:

String Input	Numeric Output	Output Class
`'500'`	500	1-by-1 scalar `double`
`'500 250 125 67'`	500, 250, 125, 67	1-by-4 row vector of `double`
`'500; 250; 125; 62.5'`	500.0000 250.0000 125.0000 62.5000	4-by-1 column vector of `double`

String Input	Numeric Output	Output Class
'1 23 6 21; 53:56'	1 23 6 21 53 54 55 56	2-by-5 matrix of double
'12e-3 5.9e-3'	0.0120 0.0059	vector of double
'uint16(500)'	500	16–bit unsigned integer

Note str2num does not operate on cell arrays. To convert a cell array of strings to a numeric value, use str2double.

If the input string does not represent a valid number or matrix, str2num(str) returns the empty matrix in x.

[x, status] = str2num('str') returns the status of the conversion in logical status, where status equals logical 1 (true) if the conversion succeeds, and logical 0 (false) otherwise.

Space characters can be significant. For instance, str2num('1+2i') and str2num('1 + 2i') produce x = 1+2i, while str2num('1 +2i') produces x = [1 2i]. You can avoid these problems by using the str2double function.

Examples

Input a character string that contains a single number. The output is a scalar double:

```
A = str2num('500')
A =
   500

class(A)
ans =
   double
```

Repeat this operation, but this time using an unsigned 16–bit integer:

```
A = str2num('uint16(500)')
A =
    500

class(A)
ans =
   uint16
```

Try three different ways of specifying a row vector. Each returns the same answer:

```
str2num('2 4 6 8')              % Separate with spaces.
ans =
     2     4     6     8

str2num('2,4,6,8')              % Separate with commas.
ans =
     2     4     6     8

str2num('[2 4 6 8]')            % Enclose in brackets.
ans =
     2     4     6     8
```

Note that the first two of these commands do not need the MATLAB square bracket operator to create a matrix. The `str2num` function inserts the brackets for you if they are needed.

Use a column vector this time:

```
str2num('2; 4; 6; 8')
ans =
     2
     4
     6
     8
```

And now a 2-by-2 matrix:

```
str2num('2 4; 6 8')
ans =
     2     4
     6     8
```

See Also

`num2str` | `str2double` | `hex2num` | `sscanf` | `sparse` | `char` | `cast` | special characters

Purpose

Concatenate strings horizontally

Syntax

```
combinedStr = strcat(s1, s2, ..., sN)
```

Description

`combinedStr = strcat(s1, s2, ..., sN)` horizontally concatenates strings in arrays `s1`, `s2`, ..., `sN`. Inputs can be combinations of single strings, strings in scalar cells, character arrays with the same number of rows, and same-sized cell arrays of strings. If any input is a cell array, `combinedStr` is a cell array of strings. Otherwise, `combinedStr` is a character array.

Tips

- For character array inputs, `strcat` removes trailing ASCII white-space characters: space, tab, vertical tab, newline, carriage return, and form-feed. To preserve trailing spaces when concatenating character arrays, use horizontal array concatenation, `[s1, s2, ..., sN]`. See the final example in the following section.
- For cell array inputs, `strcat` does not remove trailing white space.
- When combining nonscalar cell arrays and multi-row character arrays, cell arrays must be column vectors with the same number of rows as the character arrays.

Examples

Concatenate two cell arrays:

```
a = {'abcde', 'fghi'};
b = {'jkl',   'mn'};

ab = strcat(a, b)
```

MATLAB returns

```
ab =
    'abcdejkl'    'fghimn'
```

Combine cell arrays `a` and `b` from the previous example with a scalar cell:

```
c = {'Q'};
abc = strcat(a, b, c)
```

MATLAB returns

```
abc =
    'abcdejklQ'    'fghimnQ'
```

Compare the use of `strcat` and horizontal array concatenation with strings that contain trailing spaces:

```
a = 'hello  ';
b = 'goodbye';

using_strcat = strcat(a, b)
using_arrayop = [a, b]      % Equivalent to horzcat(a, b)
```

MATLAB returns

```
using_strcat =
hellogoodbye

using_arrayop =
hello  goodbye
```

See Also

cat | vertcat | horzcat | cellstr | strjoin | special character

strcmp

Purpose

Compare strings (case sensitive)

Syntax

```
TF = strcmp(string,string)
TF = strcmp(string,cellstr)
TF = strcmp(cellstr,cellstr)
```

Description

`TF = strcmp(string,string)` compares two strings for equality. The strings are considered to be equal if the size and content of each are the same. The function returns a scalar logical 1 for equality, or scalar logical 0 for inequality.

`TF = strcmp(string,cellstr)` compares a string with each element of a cell array of strings. The function returns a logical array the same size as the `cellstr` input in which logical 1 represents equality. The order of the input arguments is not important.

`TF = strcmp(cellstr,cellstr)` compares each element of one cell array of strings with the same element of the other. The function returns a logical array the same size as either cell array input.

Tips

- The `strcmp` function is intended for comparison of character data. When used to compare numeric data, it returns logical `0`.
- Use `strcmpi` for case-insensitive string comparisons.
- Any leading and trailing blanks in either of the strings are explicitly included in the comparison.
- The value returned by `strcmp` is not the same as the C language convention.
- `strcmp` supports international character sets.

Input Arguments

string

A single character string or n-by-1 array of strings.

cellstr

A cell array of strings.

Output Arguments

TF

When both inputs are character arrays, TF is a scalar logical 1 or 0. This value is logical 1 (true) if the size and content of both arrays are equal, and logical 0 (false) if it is not.

When either or both inputs is a cell array of strings, TF is an array of logical ones and zeros. This array is the same size as the input cell array(s), and contains logical 1 (true) for those elements of the input arrays that are a match, and logical 0 (false) for those elements that are not.

Examples

Perform a simple comparison of two strings:

```
strcmp('Yes', 'No')
ans =
     0
strcmp('Yes', 'Yes')
ans =
     1
```

Create two cell arrays of strings and call strcmp to compare them:

```
A = {'Handle Graphics', 'Statistics';   ...
     '  Toolboxes', 'MathWorks'};

B = {'Handle Graphics', 'Signal Processing';    ...
     'Toolboxes', 'MATHWORKS'};

match = strcmp(A, B)
match =
     1     0
     0     0
```

The result of comparing the two cell arrays is:

- match{1,1} is 1 because “Handle Graphics” in A{1,1} matches the same text in B{1,1}.

- `match{1,2}` is 0 because "Statistics" in `A{1,2}` does not match "Signal Processing" in `B{1,2}`.
- `match{2,1}` is 0 because " Toolboxes", in `A{2,1}` contains leading space characters that are not in `B{2,1}`.
- `match{2,2}` is 0 because "MathWorks" in `A{2,2}` uses different letter case than "MATHWORKS" in `B{2,2}`, and `strcmp` does a case-sensitive comparison.

The following example has three parts. It compares

- A string to an array of strings.
- A padded string to a cell array of strings.
- An unpadded string to a cell array of strings.

Start by creating the necessary data structures.

1 Cell array of strings –

Create a 3-element cell array of strings:

```
cellarr = { ...
   'There are 10 kinds of people in the world,'; ...
   'those who understand binary math,'; ...
   'and those who don''t.'};
```

2 String array –

From the cell array, create a string array. The string array contains space characters at the end of rows 2 and 3 for the padding needed to make the array rectangular:

```
strarr = char(cellarr)
strarr =
There are 10 kinds of people in the world,
those who understand binary math,
and those who don't.
```

```
%                        Each line ends here   ^
```

3 String vector –

From row 2 of the string array, create a string vector. This string is also padded with spaces at the end:

```
strvec = strarr(2,:)
strvec =
those who understand binary math,
%                     Padded line ends here   ^
```

Begin the comparisons. Start by comparing the string vector with the string array. When comparing character arrays, `strcmp` does not do a row-by-row comparison. It compares *all* of the 1-by-42 `strvec` with *all* of the 3-by-42 `strarr`. Finding them to be different, the answer is `false` and `strcmp` returns logical 0:

```
strcmp(strvec, strarr)
ans =
     0
```

Compare the string vector to the cell array. Even though `strvec` is essentially the same as row 2 of `cellarr`, it is not a match because of the space padding in `strvec`:

```
strcmp(strvec, cellarr)
ans =
     0
     0
     0
```

Remove the space padding from the string vector and compare it to the cell array. `strcmp` compares `strvec` with each row of the `cellarr`, finding a match with the second row of the latter:

```
strcmp(deblank(strvec), cellarr)
ans =
```

```
0
1
0
```

See Also

strncmp | strcmpi | strncmpi | strfind | regexp | regexpi |

Purpose

Compare strings (case insensitive)

Syntax

```
TF = strcmpi(string,string)
TF = strcmpi(string,cellstr)
TF = strcmpi(cellstr,cellstr)
```

Description

`TF = strcmpi(string,string)` compares two strings for equality, ignoring any differences in letter case. The strings are considered to be equal if the size and content of each are the same. The function returns a scalar logical 1 for equality, or scalar logical 0 for inequality.

`TF = strcmpi(string,cellstr)` compares a string with each element of a cell array of strings, ignoring letter case. The function returns a logical array the same size as the `cellstr` input in which logical 1 represents equality. The order of the input arguments is not important.

`TF = strcmpi(cellstr,cellstr)` compares each element of one cell array of strings with the same element of the other, ignoring letter case. The function returns a logical array the same size as the input arrays.

Tips

- The `strcmpi` function is intended for comparison of character data. When used to compare numeric data, it returns logical `0`.
- Use `strcmp` for case-sensitive string comparisons.
- Any leading and trailing blanks in either of the strings are explicitly included in the comparison.
- The value returned by `strcmpi` is not the same as the C language convention.
- `strcmpi` supports international character sets.

Input Arguments

string

A single character string or n-by-1 array of strings.

cellstr

A cell array of strings.

Output Arguments

TF

When both inputs are character arrays, TF is a scalar logical value. This value is logical 1 (true) if the size and content of both arrays are equal, and logical 0 (false) if they are not.

When either or both inputs are a cell array of strings, TF is an array of logical ones and zeros. This array is the same size as the input cell array(s), and contains logical 1 (true) for those elements of the input arrays that are a match, and logical 0 (false) for those elements that are not.

Examples

Perform a simple case-insensitive comparison of two strings:

```
strcmpi('Yes', 'No')
ans =
     0
strcmpi('Yes', 'yes')
ans =
     1
```

Create two cell arrays of strings and call strcmpi to compare them:

```
A = {'Handle Graphics', 'Statistics';   ...
     '  Toolboxes', 'MathWorks'};

B = {'Handle Graphics', 'Signal Processing';    ...
     'Toolboxes', 'MATHWORKS'};

match = strcmpi(A, B)
match =
     1     0
     0     1
```

The result of comparing the two cell arrays is:

- match{1,1} is 1 because “Handle Graphics” in A{1,1} matches the same text in B{1,1}.

- `match{1,2}` is 0 because “Statistics” in `A{1,2}` does not match “Signal Processing” in `B{1,2}`.
- `match{2,1}` is 0 because “ Toolboxes”, in `A{2,1}` contains leading space characters that are not in `B{2,1}`.
- `match{2,2}` is 1 because even though “MathWorks” in `A{2,2}` uses different letter case than “MATHWORKS” in `B{2,2}`, `strcmpi` performs the comparison without sensitivity to letter case.

See Also

`strncmpi` | `strcmp` | `strncmp` | `strfind` | `regexpi` | `regexp` |

stream2

Purpose

Compute 2-D streamline data

Syntax

```
XY = stream2(x,y,u,v,startx,starty)
XY = stream2(u,v,startx,starty)
XY = stream2(...,options)
```

Description

`XY = stream2(x,y,u,v,startx,starty)` computes streamlines from vector data `u` and `v`.

The arrays `x` and `y`, which define the coordinates for `u` and `v`, must be monotonic, but do not need to be uniformly spaced. `x` and `y` must have the same number of elements, as if produced by `meshgrid`.

`startx` and `starty` define the starting positions of the streamlines. The section "Specifying Starting Points for Stream Plots" provides more information on defining starting points.

The returned value `XY` contains a cell array of vertex arrays.

`XY = stream2(u,v,startx,starty)` assumes the arrays `x` and `y` are defined as `[x,y] = meshgrid(1:n,1:m)` where `[m,n] = size(u)`.

`XY = stream2(...,options)` specifies the options used when creating the streamlines. Define `options` as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:

```
[stepsize]
```

or

```
[stepsize, max_number_vertices]
```

If you do not specify a value, MATLAB software uses the default:

- Step size = 0.1 (one tenth of a cell)
- Maximum number of vertices = 10000

Use the `streamline` command to plot the data returned by `stream2`.

Examples

This example draws 2-D streamlines from data representing air currents over regions of North America.

```
load wind
[sx,sy] = meshgrid(80,20:10:50);
streamline(stream2(x(:,:,5),y(:,:,5),u(:,:,5),v(:,:,5),sx,sy));
```

See Also

coneplot | stream3 | streamline

How To

- Specifying Starting Points for Stream Plots

Purpose

Compute 3-D streamline data

Syntax

```
XYZ = stream3(X,Y,Z,U,V,W,startx,starty,startz)
XYZ = stream3(U,V,W,startx,starty,startz)
XYZ = stream3(...,options)
```

Description

`XYZ = stream3(X,Y,Z,U,V,W,startx,starty,startz)` computes streamlines from vector data `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`.

`startx`, `starty`, and `startz` define the starting positions of the streamlines. The section "Specifying Starting Points for Stream Plots" provides more information on defining starting points.

The returned value `XYZ` contains a cell array of vertex arrays.

`XYZ = stream3(U,V,W,startx,starty,startz)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:N,1:M,1:P)` where `[M,N,P] = size(U)`.

`XYZ = stream3(...,options)` specifies the options used when creating the streamlines. Define `options` as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:

```
[stepsize]
```

or

```
[stepsize, max_number_vertices]
```

If you do not specify values, MATLAB software uses the default:

- Step size = 0.1 (one tenth of a cell)
- Maximum number of vertices = 10000

Use the `streamline` command to plot the data returned by `stream3`.

Examples

This example draws 3-D streamlines from data representing air currents over regions of North America.

```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
streamline(stream3(x,y,z,u,v,w,sx,sy,sz))
view(3)
```

See Also

`coneplot` | `stream2` | `streamline`

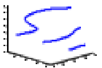
How To

- Specifying Starting Points for Stream Plots

streamline

Purpose

Plot streamlines from 2-D or 3-D vector data



Syntax

```
streamline(X,Y,Z,U,V,W,startx,starty,startz)
streamline(U,V,W,startx,starty,startz)
streamline(XYZ)
streamline(X,Y,U,V,startx,starty)
streamline(U,V,startx,starty)
streamline(XY)
streamline(...,options)
streamline(axes_handle,...)
h = streamline(...)
```

Description

`streamline(X,Y,Z,U,V,W,startx,starty,startz)` draws streamlines from 3-D vector data `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`.

`startx`, `starty`, `startz` define the starting positions of the streamlines. The section Specifying Starting Points for Stream Plots provides more information on defining starting points.

`streamline(U,V,W,startx,starty,startz)` assumes the arrays `X`, `Y`, and `Z` are defined as `[X,Y,Z] = meshgrid(1:N,1:M,1:P)`, where `[M,N,P] = size(U)`.

`streamline(XYZ)` assumes `XYZ` is a precomputed cell array of vertex arrays (as produced by `stream3`).

`streamline(X,Y,U,V,startx,starty)` draws streamlines from 2-D vector data `U`, `V`.

The arrays `X` and `Y`, which define the coordinates for `U` and `V`, must be monotonic, but do not need to be uniformly spaced. `X` and `Y` must have the same number of elements, as if produced by `meshgrid`.

startx and starty define the starting positions of the streamlines. The output argument h contains a vector of line handles, one handle for each streamline.

streamline(U,V,startx,starty) assumes the arrays X and Y are defined as [X,Y] = meshgrid(1:N,1:M), where [M,N] = size(U).

streamline(XY) assumes XY is a precomputed cell array of vertex arrays (as produced by stream2).

streamline(...,options) specifies the options used when creating the streamlines. Define options as a one- or two-element vector containing the step size or the step size and the maximum number of vertices in a streamline:

```
[stepsize]
```

or

```
[stepsize, max_number_vertices]
```

If you do not specify values, MATLAB uses the default:

- Step size = 0.1 (one tenth of a cell)
- Maximum number of vertices = 1000

streamline(axes_handle,...) plots into the axes object with the handle axes_handle instead of the into current axes object (gca).

h = streamline(...) returns a vector of line handles, one handle for each streamline.

Examples

This example draws streamlines from data representing air currents over a region of North America. Loading the wind data set creates the variables x, y, z, u, v, and w in the MATLAB workspace.

The plane of streamlines indicates the flow of air from the west to the east (the *x*-direction) beginning at x = 80 (which is close to the minimum value of the x coordinates). The *y*- and *z*-coordinate starting points are multivalued and approximately span the range of

these coordinates. meshgrid generates the starting positions of the streamlines.

```
load wind
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
h = streamline(x,y,z,u,v,w,sx,sy,sz);
set(h,'Color','red')
view(3)
```

See Also

coneplot | stream2 | stream3 | streamparticles | meshgrid

How To

- Specifying Starting Points for Stream Plots
- Stream Line Plots of Vector Data

Purpose

Plot stream particles

Syntax

```
streamparticles(vertices)
streamparticles(vertices,n)
streamparticles(...,'PropertyName',PropertyValue,...)
streamparticles(line_handle,...)
h = streamparticles(...)
```

Description

`streamparticles(vertices)` draws stream particles of a vector field. Stream particles are usually represented by markers and can show the position and velocity of a streamline. `vertices` is a cell array of 2-D or 3-D vertices (as if produced by `stream2` or `stream3`).

`streamparticles(vertices,n)` uses `n` to determine how many stream particles to draw. The `ParticleAlignment` property controls how `n` is interpreted.

- If `ParticleAlignment` is set to `off` (the default) and `n` is greater than `1`, approximately `n` particles are drawn evenly spaced over the streamline vertices.

  If `n` is less than or equal to `1`, `n` is interpreted as a fraction of the original stream vertices; for example, if `n` is `0.2`, approximately 20% of the vertices are used.

  `n` determines the upper bound for the number of particles drawn. The actual number of particles can deviate from `n` by as much as a factor of 2.

- If `ParticleAlignment` is `on`, `n` determines the number of particles on the streamline having the most vertices and sets the spacing on the other streamlines to this value. The default value is `n = 1`.

`streamparticles(...,'`*`PropertyName`*`',PropertyValue,...)` controls the stream particles using named properties and specified values. Any unspecified properties have default values. MATLAB ignores the case of property names.

Stream Particle Properties

`Animate` — Stream particle motion [nonnegative integer]

The number of times to animate the stream particles. The default is 0, which does not animate. Inf animates until you enter **Ctrl+C**.

FrameRate — Animation frames per second [nonnegative integer]

This property specifies the number of frames per second for the animation. Inf, the default, draws the animation as fast as possible. Note that the speed of the animation might be limited by the speed of the computer. In such cases, the value of FrameRate cannot necessarily be achieved.

ParticleAlignment — Align particles with streamlines [on | {off}]

Set this property to on to draw particles at the beginning of each streamline. This property controls how streamparticles interprets the argument n (number of stream particles).

Stream particles are line objects. In addition to stream particle properties, you can specify any line object property, such as Marker and EraseMode. streamparticles sets the following line properties when called.

Line Property	Value Set by streamparticles
EraseMode	xor
LineStyle	none
Marker	o
MarkerEdgeColor	none
MarkerFaceColor	red

You can override any of these properties by specifying a property name and value as arguments to streamparticles. For example, this statement uses RGB values to set the MarkerFaceColor to medium gray:

```
streamparticles(vertices,'MarkerFaceColor',[.5 .5 .5])
```

`streamparticles(line_handle,...)` uses the line object identified by `line_handle` to draw the stream particles.

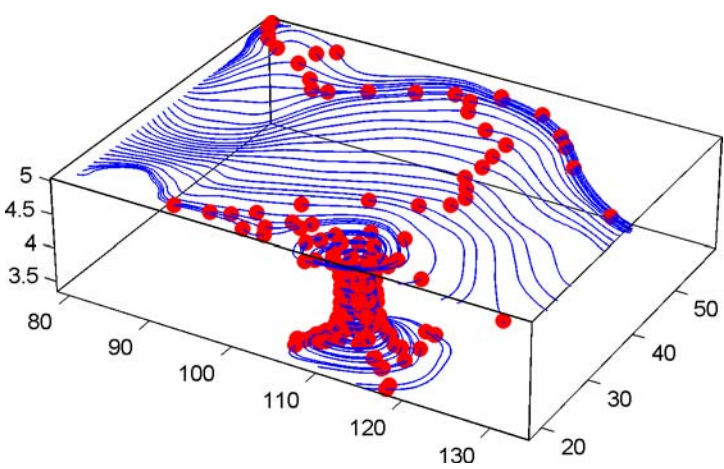
`h = streamparticles(...)` returns a vector of handles to the line objects it creates.

Examples

This example combines streamlines with stream particle animation. The `interpstreamspeed` function determines the vertices along the streamlines where stream particles will be drawn during the animation, thereby controlling the speed of the animation. Setting the axes `DrawMode` property to `fast` provides faster rendering.

```
load wind
[sx sy sz] = meshgrid(80,20:1:55,5);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
sl = streamline(verts);
iverts = interpstreamspeed(x,y,z,u,v,w,verts,.025);
axis tight; view(30,30); daspect([1 1 .125])
camproj perspective; camva(8)
set(gca,'DrawMode','fast')
box on
streamparticles(iverts,35,'animate',10,'ParticleAlignment','on')
```

The following picture is a static view of the animation.



This example uses the streamlines in the `z = 5` plane to animate the flow along these lines with `streamparticles`.

```
load wind
daspect([1 1 1]); view(2)
[verts averts] = streamslice(x,y,z,u,v,w,[],[],[5]);
sl = streamline([verts averts]);
axis tight off;
set(sl,'Visible','off')
iverts = interpstreamspeed(x,y,z,u,v,w,verts,.05);
set(gca,'DrawMode','fast','Position',[0 0 1 1],'ZLim',[4.9 5.1])
set(gcf,'Color','black')
streamparticles(iverts, 200, ...
    'Animate',100,'FrameRate',40, ...
    'MarkerSize',10,'MarkerFaceColor','yellow')
```

See Also

`interpstreamspeed` | `stream3` | `streamline` | `stream2`

How To

- Creating Stream Particle Animations
- Specifying Starting Points for Stream Plots

Purpose

3-D stream ribbon plot from vector volume data

Syntax

```
streamribbon(X,Y,Z,U,V,W,startx,starty,startz)
streamribbon(U,V,W,startx,starty,startz)
streamribbon(vertices,X,Y,Z,cav,speed)
streamribbon(vertices,cav,speed)
streamribbon(vertices,twistangle)
streamribbon(...,width)
streamribbon(axes_handle,...)
h = streamribbon(...)
```

Description

`streamribbon(X,Y,Z,U,V,W,startx,starty,startz)` draws stream ribbons from vector volume data `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`.

`startx`, `starty`, and `startz` define the starting positions of the stream ribbons at the center of the ribbons. The section Specifying Starting Points for Stream Plots provides more information on defining starting points.

The twist of the ribbons is proportional to the curl of the vector field. The width of the ribbons is calculated automatically.

`streamribbon(U,V,W,startx,starty,startz)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamribbon(vertices,X,Y,Z,cav,speed)` assumes precomputed streamline vertices, curl angular velocity, and flow speed. `vertices` is a

cell array of streamline vertices (as produced by `stream3`). `X`, `Y`, `Z`, `cav`, and `speed` are 3-D arrays.

`streamribbon(vertices,cav,speed)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(cav)`.

`streamribbon(vertices,twistangle)` uses the cell array of vectors `twistangle` for the twist of the ribbons (in radians). The size of each corresponding element of `vertices` and `twistangle` must be equal.

`streamribbon(...,width)` sets the width of the ribbons to `width`.

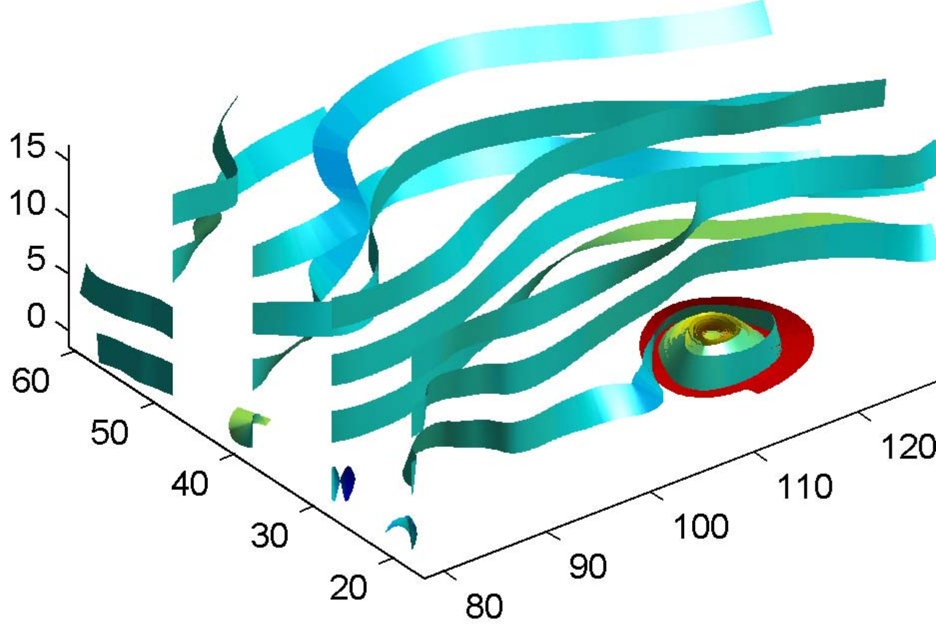
`streamribbon(axes_handle,...)` plots into the axes object with the handle `axes_handle` instead of into the current axes object (`gca`).

`h = streamribbon(...)` returns a vector of handles (one per start point) to `surface` objects.

Examples

This example uses stream ribbons to indicate the flow in the `wind` data set. Inputs include the coordinates, vector field components, and starting location for the stream ribbons.

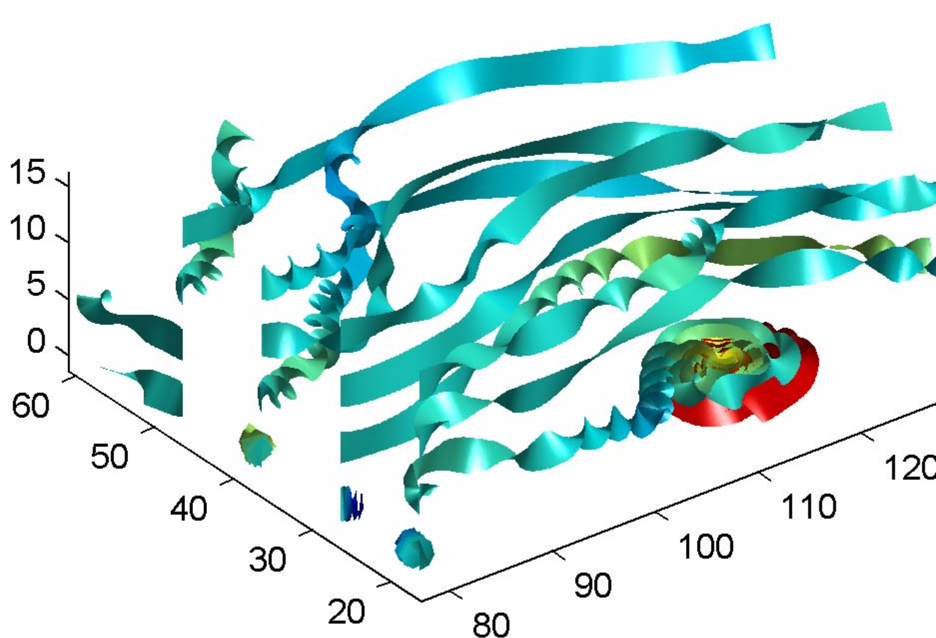
```
figure
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
streamribbon(x,y,z,u,v,w,sx,sy,sz);
% Define viewing and lighting
axis tight
shading interp;
view(3);
camlight; lighting gouraud
```



This example uses precalculated vertex data (`stream3`), curl average velocity (`curl`), and speed $\sqrt{u^2 + v^2 + w^2}$. Using precalculated data enables you to use values other than those calculated from the single data source. In this case, the speed is reduced by a factor of 10 compared to the previous example.

```
figure
```

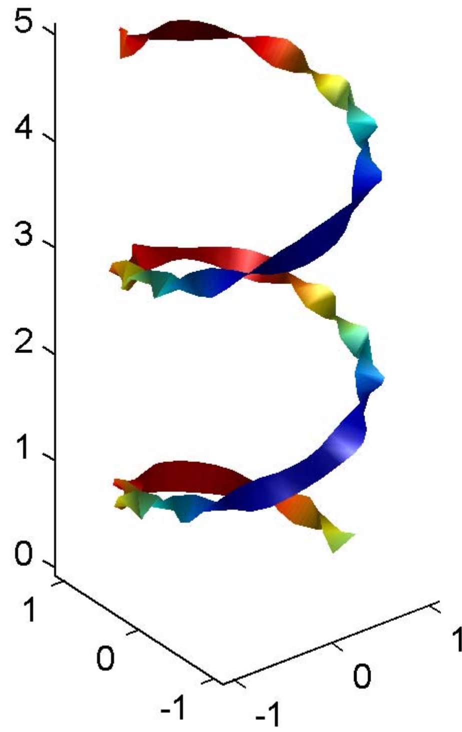
```
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
cav = curl(x,y,z,u,v,w);
spd = sqrt(u.^2 + v.^2 + w.^2).*.1;
streamribbon(verts,x,y,z,cav,spd);
% Define viewing and lighting
axis tight
shading interp
view(3)
camlight; lighting gouraud
```



This example specifies a twist angle for the stream ribbon.

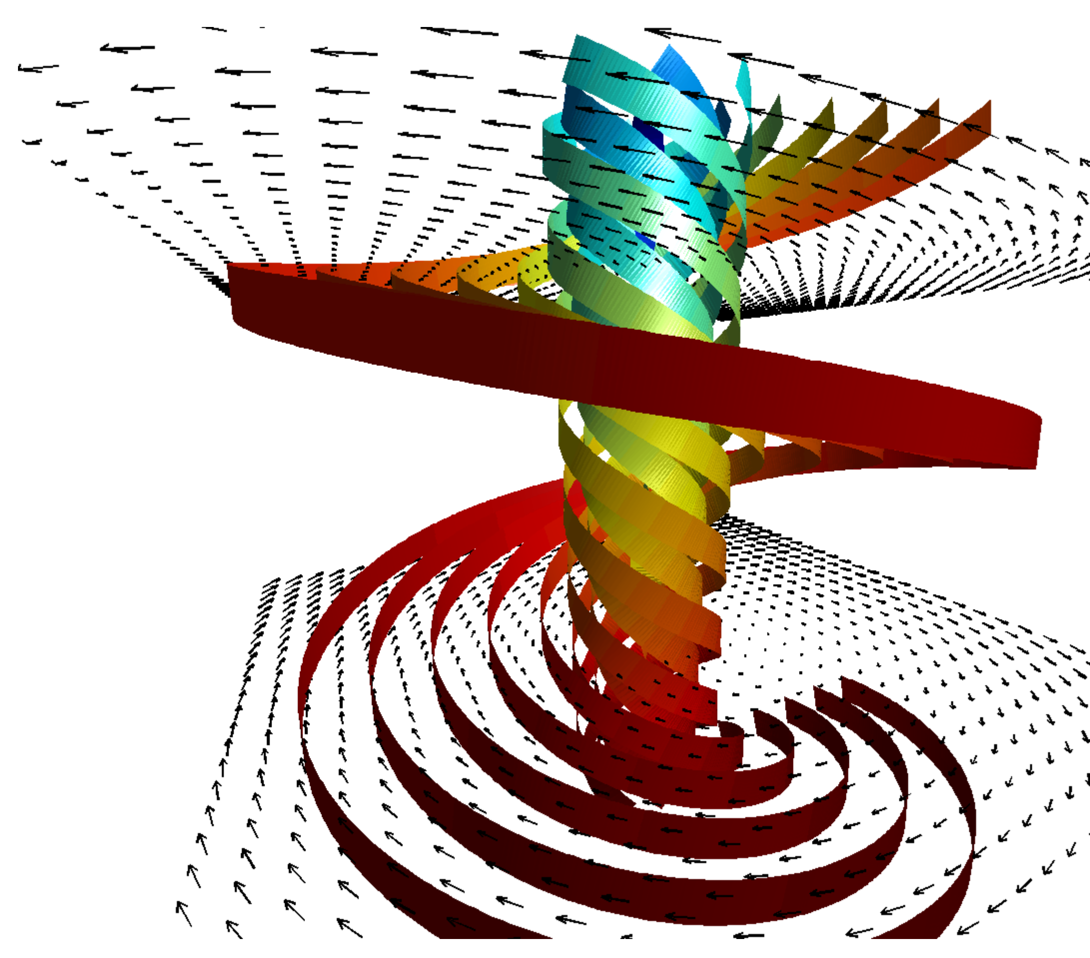
```
figure
t = 0:.15:15;
verts = {[cos(t)' sin(t)' (t/3)']};
twistangle = {cos(t)'};
streamribbon(verts,twistangle);
% Define viewing and lighting
```

```
axis tight
shading interp;
view(3);
camlight; lighting gouraud
```



This example combines cone plots (`coneplot`) and stream ribbon plots in one graph.

```
figure
% Define 3-D arrays x, y, z, u, v, w
xmin = -7; xmax = 7;
ymin = -7; ymax = 7;
zmin = -7; zmax = 7;
x = linspace(xmin,xmax,30);
y = linspace(ymin,ymax,20);
z = linspace(zmin,zmax,20);
[x y z] = meshgrid(x,y,z);
u = y; v = -x; w = 0*x+1;
[cx cy cz] = meshgrid(linspace(xmin,xmax,30),...
   linspace(ymin,ymax,30),[-3 4]);
h = coneplot(x,y,z,u,v,w,cx,cy,cz,'quiver');
set(h,'Color','k');

% Plot two sets of streamribbons
[sx sy sz] = meshgrid([-1 0 1],[-1 0 1],-6);
streamribbon(x,y,z,u,v,w,sx,sy,sz);
[sx sy sz] = meshgrid([1:6],[0],-6);
streamribbon(x,y,z,u,v,w,sx,sy,sz);

% Define viewing and lighting
shading interp
view(-30,10) ; axis off tight
camproj perspective; camva(66); camlookat;
camdolly(0,0,.5,'fixtarget')
camlight
```



See Also

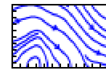
`curl` | `streamtube` | `streamline` | `stream3` | `meshgrid` | `coneplot`

How To

- "Volume Visualization"
- Displaying Curl with Stream Ribbons
- Specifying Starting Points for Stream Plots

Purpose

Plot streamlines in slice planes



Syntax

```
streamslice(X,Y,Z,U,V,W,startx,starty,startz)
streamslice(U,V,W,startx,starty,startz)
streamslice(X,Y,U,V)
streamslice(U,V)
streamslice(...,density)
streamslice(...,'arrowsmode')
streamslice(...,'method')
streamslice(axes_handle,...)
h = streamslice(...)
[vertices arrowvertices] = streamslice(...)
```

Description

`streamslice(X,Y,Z,U,V,W,startx,starty,startz)` draws well-spaced streamlines (with direction arrows) from vector data `U`, `V`, `W` in axis aligned *x*-, *y*-, *z*-planes starting at the points in the vectors `startx`, `starty`, `startz`. (The section Specifying Starting Points for Stream Plots provides more information on defining starting points.)

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`. `U`, `V`, `W` must be m-by-n-by-p volume arrays.

Do not assume that the flow is parallel to the slice plane. For example, in a stream slice at a constant `z`, the `z` component of the vector field `W` is ignored when you are calculating the streamlines for that plane.

Stream slices are useful for determining where to start streamlines, stream tubes, and stream ribbons.

`streamslice(U,V,W,startx,starty,startz)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where [m,n,p] = size(U).

streamslice(X,Y,U,V) draws well-spaced streamlines (with direction arrows) from vector volume data U, V.

The arrays X and Y, which define the coordinates for U and V, must be monotonic, but do not need to be uniformly spaced. X and Y must have the same number of elements, as if produced by meshgrid.

streamslice(U,V) assumes X, Y, and Z are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where [m,n,p] = size(U).

streamslice(...,density) modifies the automatic spacing of the streamlines. density must be greater than 0. The default value is 1; higher values produce more streamlines on each plane. For example, 2 produces approximately twice as many streamlines, while 0.5 produces approximately half as many.

streamslice(...,'*arrowsmode*') determines if direction arrows are present or not. arrowmode can be

- arrows — Draw direction arrows on the streamlines (default).
- noarrows — Do not draw direction arrows.

streamslice(...,'*method*') specifies the interpolation method to use. method can be

- linear — Linear interpolation (default)
- cubic — Cubic interpolation
- nearest — Nearest-neighbor interpolation

See interp3 for more information on interpolation methods.

streamslice(axes_handle,...) plots into the axes object with the handle axes_handle instead of into the current axes object (gca).

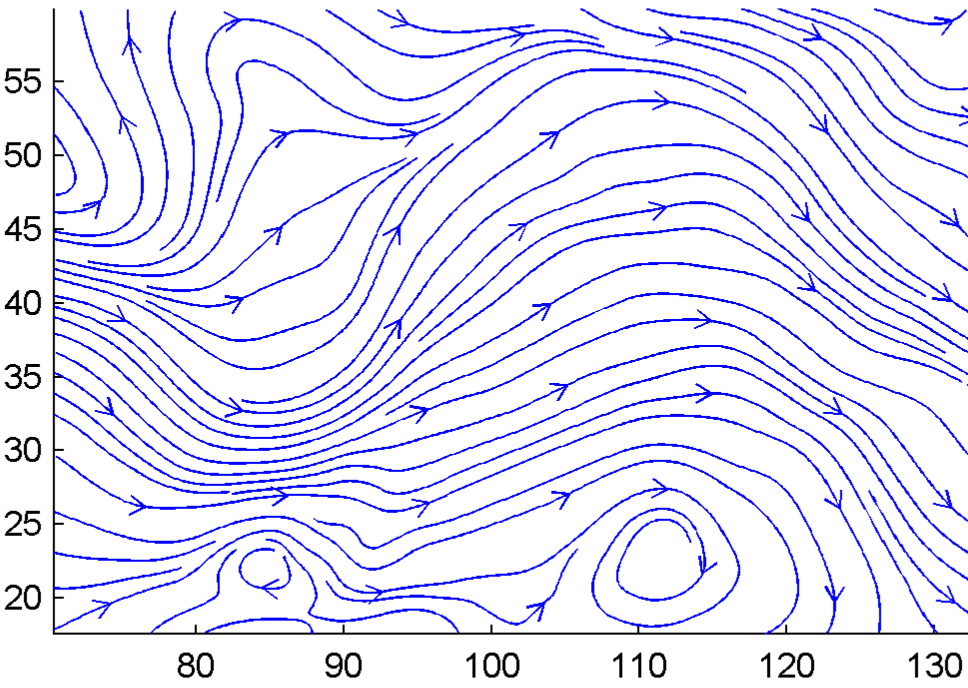
`h = streamslice(...)` returns a vector of handles to the line objects created.

`[vertices arrowvertices] = streamslice(...)` returns two cell arrays of vertices for drawing the streamlines and the arrows. You can pass these values to any of the streamline drawing functions (`streamline`, `streamribbon`, `streamtube`).

Examples

This example creates a stream slice in the `wind` data set at `z = 5`.

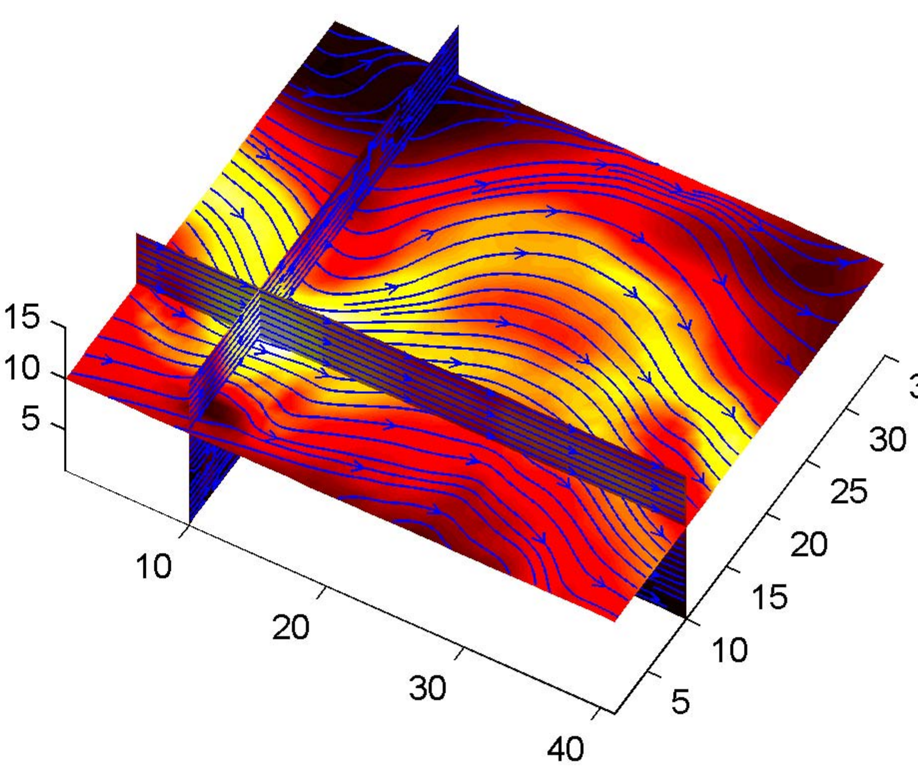
```
figure
load wind
streamslice(x,y,z,u,v,w,[],[],[5])
axis tight
```



This example uses `streamslice` to calculate vertex data for the streamlines and the direction arrows. This data is then used by `streamline` to plot the lines and arrows. Slice planes illustrating with color the wind speed $\sqrt{u^2 + v^2 + w^2}$ are drawn by `slice` in the same planes.

```
figure
```

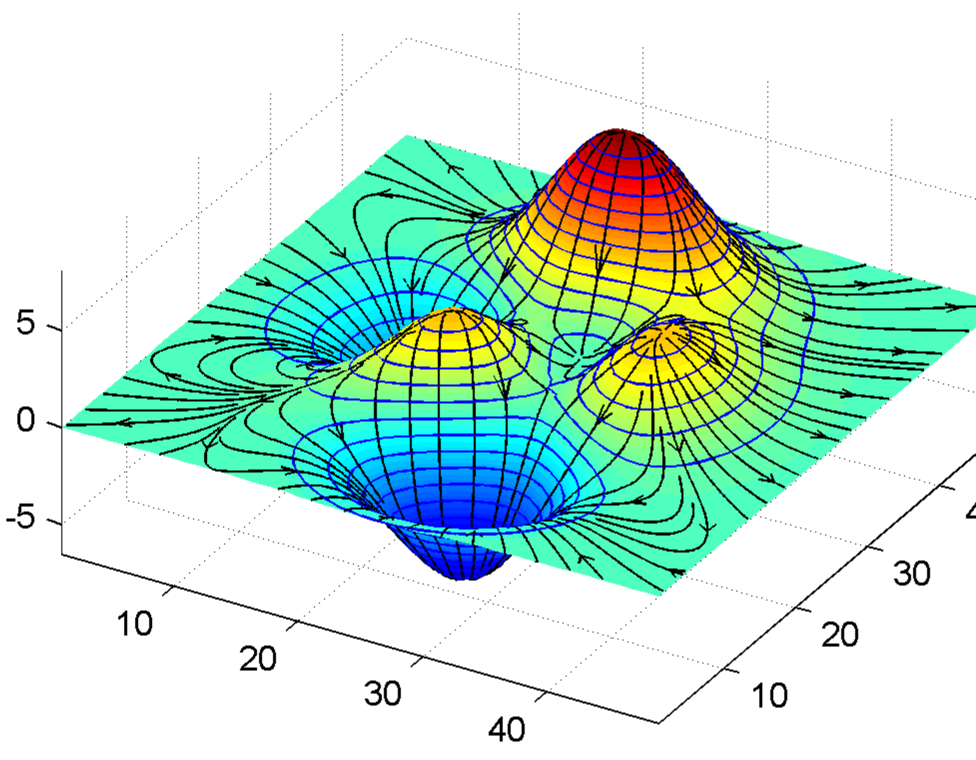
```
load wind
[verts averts] = streamslice(u,v,w,10,10,10);
streamline([verts averts])
spd = sqrt(u.^2 + v.^2 + w.^2);
hold on;
slice(spd,10,10,10);
colormap(hot)
shading interp
view(30,50); axis(volumebounds(spd));
camlight; material([.5 1 0])
```



This example superimposes contour lines on a surface and then uses `streamslice` to draw lines that indicate the gradient of the surface. `interp2` is used to find the points for the lines that lie on the surface.

```
figure
z = peaks;
surf(z)
shading interp
```

```
hold on
[c ch] = contour3(z,20); set(ch,'edgecolor','b')
[u v] = gradient(z);
h = streamslice(-u,-v);
set(h,'color','k')
for i=1:length(h);
 zi = interp2(z,get(h(i),'xdata'),get(h(i),'ydata'));
 set(h(i),'zdata',zi);
end
view(30,50); axis tight
```



See Also

contourslice | slice | streamline | volumebounds | meshgrid | interp3 | interp2 | streamribbon | streamtube

How To

- Specifying Starting Points for Stream Plots

Purpose

Create 3-D stream tube plot

Syntax

```
streamtube(X,Y,Z,U,V,W,startx,starty,startz)
streamtube(U,V,W,startx,starty,startz)
streamtube(vertices,X,Y,Z,divergence)
streamtube(vertices,divergence)
streamtube(vertices,width)
streamtube(vertices)
streamtube(...,[scale n])
streamtube(axes_handle,...)
h = streamtube(...z)
```

Description

`streamtube(X,Y,Z,U,V,W,startx,starty,startz)` draws stream tubes from vector volume data `U`, `V`, `W`.

The arrays `X`, `Y`, and `Z`, which define the coordinates for `U`, `V`, and `W`, must be monotonic, but do not need to be uniformly spaced. `X`, `Y`, and `Z` must have the same number of elements, as if produced by `meshgrid`.

`startx`, `starty`, and `startz` define the starting positions of the streamlines at the center of the tubes. The section Specifying Starting Points for Stream Plots provides more information on defining starting points.

The width of the tubes is proportional to the normalized divergence of the vector field.

`streamtube(U,V,W,startx,starty,startz)` assumes `X`, `Y`, and `Z` are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where `[m,n,p] = size(U)`.

`streamtube(vertices,X,Y,Z,divergence)` assumes precomputed streamline vertices and divergence. `vertices` is a cell array of

streamline vertices (as produced by stream3). X, Y, Z, and divergence are 3-D arrays.

streamtube(vertices,divergence) assumes X, Y, and Z are determined by the expression

```
[X,Y,Z] = meshgrid(1:n,1:m,1:p)
```

where [m,n,p] = size(divergence).

streamtube(vertices,width) specifies the width of the tubes in the cell array of vectors, width. The size of each corresponding element of vertices and width must be equal. width can also be a scalar, specifying a single value for the width of all stream tubes.

streamtube(vertices) selects the width automatically.

streamtube(...,[scale n]) scales the width of the tubes by scale. The default is scale = 1. When the stream tubes are created, using start points or divergence, specifying scale = 0 suppresses automatic scaling. n is the number of points along the circumference of the tube. The default is n = 20.

streamtube(axes_handle,...) plots into the axes object with the handle axes_handle instead of into the current axes object (gca).

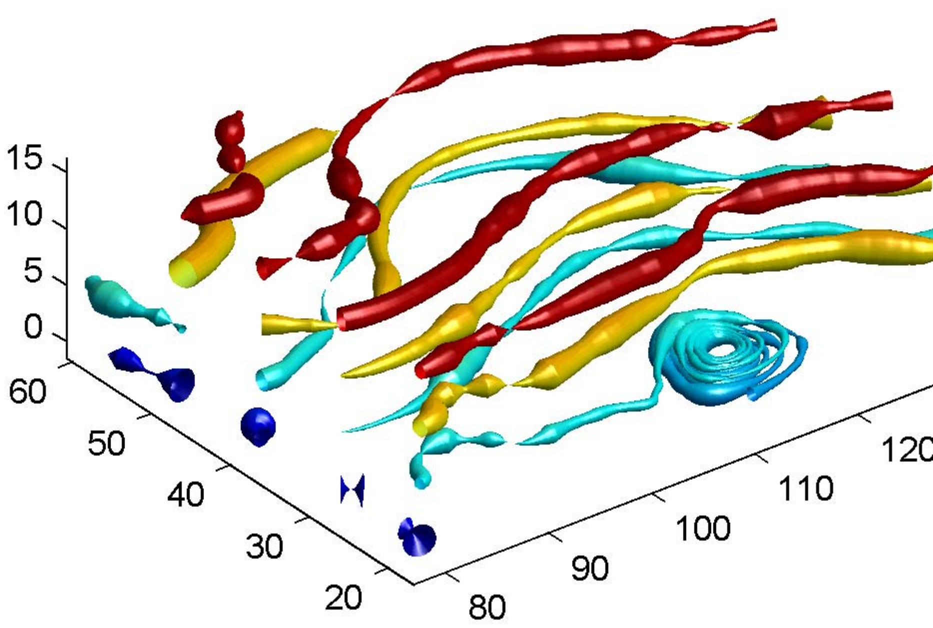
h = streamtube(...z) returns a vector of handles (one per start point) to surface objects used to draw the stream tubes.

Examples

This example uses stream tubes to indicate the flow in the wind data set. Inputs include the coordinates, vector field components, and starting location for the stream tubes.

```
figure
load wind
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
streamtube(x,y,z,u,v,w,sx,sy,sz);
% Define viewing and lighting
view(3)
axis tight
```

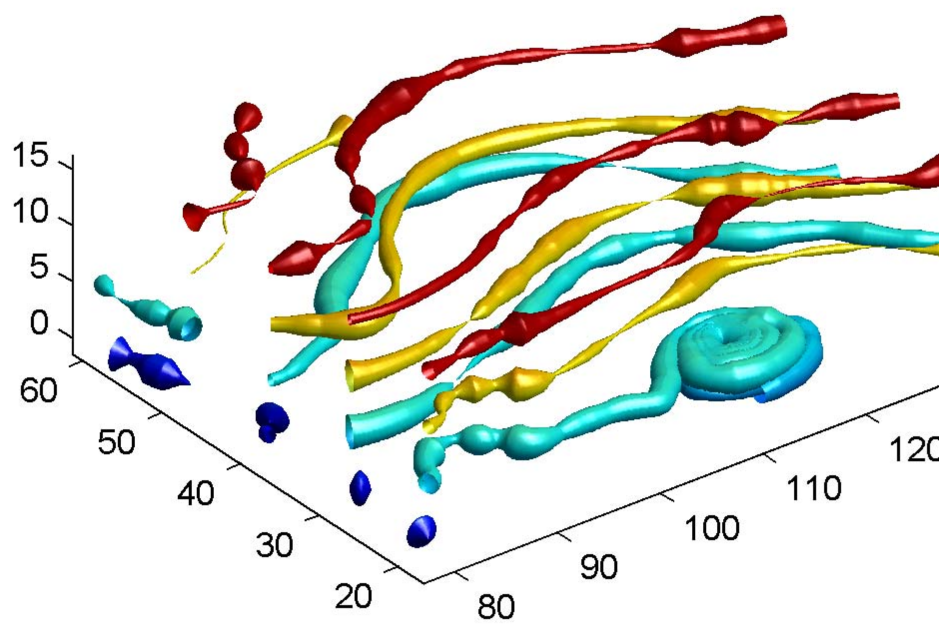
```
shading interp;
camlight; lighting gouraud
```



This example uses precalculated vertex data (`stream3`) and divergence (`divergence`).

```
figure
load wind
```

```
[sx sy sz] = meshgrid(80,20:10:50,0:5:15);
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
div = divergence(x,y,z,u,v,w);
streamtube(verts,x,y,z,-div);
% Define viewing and lighting
view(3)
axis tight
shading interp
camlight; lighting gouraud
```



See Also

`divergence` | `streamribbon` | `streamline` | `stream3` | `meshgrid` | `stream3`

How To

- Displaying Divergence with Stream Tubes
- Specifying Starting Points for Stream Plots

strfind

Purpose

Find one string within another

Syntax

```
k = strfind(str, pattern)
k = strfind(cellstr, pattern)
```

Description

`k = strfind(str, pattern)` searches the string `str` for occurrences of a shorter string, `pattern`, and returns the starting index of each such occurrence in the double array `k`. If `pattern` is not found in `str`, or if `pattern` is longer than `str`, then `strfind` returns the empty array `[]`.

`k = strfind(cellstr, pattern)` searches each string in cell array of strings `cellstr` for occurrences of a shorter string, `pattern`, and returns the starting index of each such occurrence in cell array `k`. If `pattern` is not found in a string or if `pattern` is longer then all strings in the cell array, then `strfind` returns the empty array `[]`, for that string in the cell array.

Tips

- The search performed by `strfind` is case sensitive.
- Any leading and trailing blanks in `pattern` or in the strings being searched are explicitly included in the comparison.
- The `strfind` function does not find empty strings (`''`) within a string.

Examples

Use `strfind` to find a two-letter pattern in string `S`:

```
S = 'Find the starting indices of the pattern string';
strfind(S, 'in')
ans =
     2    15    19    45

strfind(S, 'In')
ans =
     []

strfind(S, ' ')
ans =
     5     9    18    26    29    33    41
```

Use `strfind` on a cell array of strings:

```
cstr = {'How much wood would a woodchuck chuck';
        'if a woodchuck could chuck wood?'};

idx = strfind(cstr, 'wood');

idx{:,:}
ans =
    10    23
ans =
     6    28
```

This means that `'wood'` occurs at indices 10 and 23 in the first string and at indices 6 and 28 in the second.

See Also

`strtok` | `strcmp` | `strncmp` | `strcmpi` | `strncmpi` | `regexp` | `regexpi` | `regexprep` | `strrep` | `strsplit`

strings

Purpose

String handling

Syntax

```
S = 'Any Characters'
S = [S1 S2 ...]
C = {S1 S2 ...}
S = strcat(S1, S2, ...)
S = char(S1, S2, ...)
S = char(X)
X = double(S)
```

Description

`S = 'Any Characters'` creates a character array, or string. The string is actually a vector that contains the numeric codes for the characters (codes 0 to 127 are ASCII). The length of `S` is the number of characters. A quotation within the string is indicated by two quotation marks.

`S = [S1 S2 ...]` concatenates character arrays `S1`, `S2`, etc. into a new character array, `S`.

`C = {S1 S2 ...}` creates a cell array of strings. Separate each row of the cell array with a semicolon (`;`).

`S = strcat(S1, S2, ...)` horizontally concatenates `S1`, `S2`, etc., which can be character arrays or cell arrays of strings. If the inputs are character arrays, `strcat` removes trailing white space. For more information, see the `strcat` reference page.

`S = char(S1, S2, ...)` vertically concatenates character arrays `S1`, `S2`, etc., padding each input string as needed so that each row contains the same number of characters.

`S = char(X)` converts an array that contains positive integers representing numeric codes into a MATLAB character array.

`X = double(S)` converts the string to its equivalent integer numeric codes.

Tips

- To convert between character arrays and cell arrays of strings, use `char` and `cellstr`. Most string functions support both types.

- To determine whether `S` is a character array or cell array, call `ischar(S)` or `iscellstr(S)`.

Examples

Create a simple string that includes a single quote.

```
msg = 'You''re right!'

msg =
You're right!
```

Create the string `name` using two methods of concatenation.

```
name = ['Thomas' ' R. ' 'Lee']
name = strcat('Thomas',' R.',' Lee')
```

Create a character array of strings.

```
C = char('Hello','Goodbye','Yes','No')

C =
Hello
Goodbye
Yes
No
```

Create a cell array of strings.

```
S = {'Hello' 'Goodbye'; 'Yes' 'No'}

S =
    'Hello'    'Goodbye'
    'Yes'      'No'
```

See Also

`char` | `isstrprop` | `cellstr` | `ischar` | `isletter` | `isspace` | `iscellstr` | `sprintf` | `sscanf` | `text` | `input`

Concepts

- “Cell Arrays of Strings”

strjoin

Purpose

Join strings in cell array into single string

Syntax

```
str = strjoin(C)
str = strjoin(C,delimiter)
```

Description

`str = strjoin(C)` constructs the string, `str`, by linking each string in the cell array, `C`, with a single space.

`str = strjoin(C,delimiter)` constructs the string, `str`, by linking each string of `C` with the elements in `delimiter`.

Input Arguments

C - Input text

1-by-n cell array of strings

Input text, specified as a 1-by-n cell array of strings. Each element in the cell array must contain a single string in a single row.

Example: `{'The','rain','in','Spain'}`

Data Types
`cell`

delimiter - Delimiting characters

string | 1-by-n cell array of strings

Delimiting characters, specified as a single string or a 1-by-n cell array of strings.

- If `delimiter` is a single string, then `strjoin` forms `str` by inserting `delimiter` between each element of `C`. The `delimiter` input can include any of these escape sequences:

`\\`	Backslash
`\0`	Null
`\a`	Alarm
`\b`	Backspace

`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab

- If `delimiter` is a cell array of strings, then it must contain one fewer element than `C`. Each element in the cell array must contain a single string in a single row. `strjoin` forms `str` by interleaving the elements of `delimiter` and `C`. All characters in `delimiter` are inserted as literal text, and escape sequences are not supported.

Example: `', '`

Example: `{',',' '}`

Data Types
`char | cell`

Examples

Join List of Words with Whitespace

Join individual strings in a cell array of strings, `C`, with a single space.

```
C = {'one','two','three'};
str = strjoin(C)

str =

one two three
```

Join Strings as Comma Separated List

```
C = {'Newton','Gauss','Euclid','Lagrange'};
str = strjoin(C,', ')

str =

Newton, Gauss, Euclid, Lagrange
```

Join Strings with Multiple Different Delimiters

Specify multiple different delimiters in a cell array of strings. The `delimiter` cell array must have one fewer element than `C`.

```
C = {'one','two','three'};
str = strjoin(C,{' + ',' = '})
```

```
str =

one + two = three
```

See Also

strsplit | strcat | cellstr | regexp

Purpose

Justify character array

Syntax

```
T = strjust(S)
T = strjust(S, 'right')
T = strjust(S, 'left')
T = strjust(S, 'center')
```

Description

`T = strjust(S)` or `T = strjust(S, 'right')` returns a right-justified version of the character array `S`.

`T = strjust(S, 'left')` returns a left-justified version of `S`.

`T = strjust(S, 'center')` returns a center-justified version of `S`.

See Also

`deblank` | `strtrim`

strmatch

Purpose

Find possible matches for string

Note `strmatch` is not recommended. Use `strncmp` or `validatestring`, depending on your requirements, instead. `strncmp` returns the numeric index of all array elements that begin with the specified string, whereas `validatestring` returns a single string that represents the best match to the specified string. See Example 2, below.

To find an exact match for a string, use `strcmp`.

Syntax

```
x = strmatch(str, strarray)
x = strmatch(str, strarray, 'exact')
```

Description

`x = strmatch(str, strarray)` looks through the rows of the character array or cell array of strings `strarray` to find strings that begin with the text contained in `str`, and returns the matching row indices. If `strmatch` does not find `str` in `strarray`, `x` is an empty matrix (`[]`). Any trailing space characters in `str` or `strarray` are ignored when matching. `strmatch` is fastest when `strarray` is a character array.

`x = strmatch(str, strarray, 'exact')` compares `str` with each row of `strarray`, looking for an exact match of the entire strings. Any trailing space characters in `str` or `strarray` are ignored when matching.

Examples

Example 1

The statement

```
x = strmatch('max', char('max', 'minimax', 'maximum'))
```

returns `x = [1; 3]` since rows 1 and 3 begin with `'max'`. The statement

```
x = strmatch('max', char('max', 'minimax', 'maximum'),'exact')
```

returns `x = 1`, since only row 1 matches `'max'` exactly.

Example 2

This example shows how to replace use of the `strmatch` function with `validatestring` or `strncmp`.

To start with, use `strmatch` to return the index of those elements for which there is a match:

```
list = {'max', 'minimax', 'maximum', 'max'}
x = strmatch('max',list)
x =
     1
     3
     4
```

`validatestring` returns the string representing the best match. If multiple or no matches exist, this statement would return an error:

```
list = {'max', 'minimax', 'maximum', 'max'};
x = validatestring('max', list)
x =
   max
```

`strncmp` returns a logical array indicating which strings match the specified string:

```
list = {'max', 'minimax', 'maximum', 'max'};
x = strncmp('max', list, 3)
x =
     1     0     1     1
```

If you prefer that MATLAB return the numeric indices of `list`, use `find` as follows:

```
list = {'max', 'minimax', 'maximum', 'max'}
x = find(strncmp(list, 'max', 3))
```

If your input to `strmatch` is a character matrix, then first convert the matrix to a cell array using `cellstr`. Then, pass the output from `cellstr` to `strncmp` or `validatestring`

See Also

`strcmp` | `strcmpi` | `strncmp` | `strncmpi` | `strfind` | `regexp` | `regexpi` | `regexprep`

Purpose

Compare first `n` characters of strings (case sensitive)

Syntax

```
TF = strncmp(string,string,n)
TF = strncmp(string,cellstr,n)
TF = strncmp(cellstr,cellstr,n)
```

Description

`TF = strncmp(string,string,n)` compares the first `n` characters of two strings for equality. The function returns a scalar logical 1 for equality, or scalar logical 0 for inequality.

`TF = strncmp(string,cellstr,n)` compares the first `n` characters of a string with the first `n` characters of each element of a cell array of strings. The function returns a logical array the same size as the `cellstr` input in which logical 1 represents equality. The order of the first two input arguments is not important.

`TF = strncmp(cellstr,cellstr,n)` compares each element of one cell array of strings with the same element of the other. `strncmp` attempts to match only the first `n` characters of these strings. The function returns a logical array the same size as either input array.

Tips

- The `strncmp` function is intended for comparison of character data. When used to compare numeric data, it returns logical `0`.
- Use `strncmpi` for case-insensitive string comparisons.
- Any leading and trailing blanks in either of the strings are explicitly included in the comparison.
- The value returned by `strncmp` is not the same as the C language convention.
- `strncmp` supports international character sets.

Input Arguments

string

Single character string or n-by-1 array of strings.

cellstr

Cell array of strings.

n

Maximum number of characters to compare. Must be a scalar, integer-valued double.

Output Arguments

TF

When both inputs are character arrays, `TF` is a scalar logical value. This value is logical 1 (`true`) if the size and content of both arrays are equal, and logical 0 (`false`) if they are not.

When either or both inputs are a cell array of strings, `TF` is an array of logical ones and zeros. This array is the same size as the input cell array(s), and contains logical 1 (`true`) for those elements of the input arrays that are a match, and logical 0 (`false`) for those elements that are not.

Examples

Before trying the `strncmp` function, use `strcmp` to perform a simple comparison of the two input strings. Because only the first 13 characters are the same, `strcmp` returns logical 0:

```
strcmp('Kansas City, KS', 'Kansas City, MO')
ans =
     0
```

Do the comparison again, but this time using `strncmp` and specifying the number of characters to compare:

```
chars2compare = length('Kansas City, KS') - 2
ans =
    13
strncmp('Kansas City, KS', 'Kansas City, MO', chars2compare)
ans =
     1
```

From a list of 10 MATLAB functions, find those that apply to using a camera:

```
function_list = {'calendar' 'case' 'camdolly' 'circshift' ...
                 'caxis' 'camtarget' 'cast' 'camorbit' ...
                 'callib' 'cart2sph'};

strncmp(function_list, 'cam', 3)
ans =
    0    0    1    0    0    1    0    1    0    0

function_list{strncmp(function_list, 'cam', 3)}
ans =
   camdolly
ans =
   camtarget
ans =
   camorbit
```

Create two 5-by-10 string arrays `str1` and `str2` that are equal, except for the element at row 4, column 3. Using linear indexing, this is element 14:

```
str1 = ['AAAAAAAAAA'; 'BBBBBBBBBB'; 'CCCCCCCCCC'; ...
        'DDDDDDDDDD'; 'EEEEEEEEEE']
str1 =
    AAAAAAAAAA
    BBBBBBBBBB
    CCCCCCCCCC
    DDDDDDDDDD
    EEEEEEEEEE

str2 = str1;
str2(4,3) = '-'
str2 =
    AAAAAAAAAA
```

```
BBBBBBBBBB
CCCCCCCCCC
DD-DDDDDDD
EEEEEEEEEE
```

Because MATLAB compares the arrays in linear order (that is, column by column rather than row by row), `strncmp` finds only the first 13 elements to be the same:

```
str1   A B C D E A B C D E A B C D E
str2   A B C D E A B C D E A B C - E
                                 |
                             element 14

strncmp(str1, str2, 13)
ans =
     1

strncmp(str1, str2, 14)
ans =
     0
```

See Also

`strcmp` | `strncmpi` | `strcmpi` | `strfind` | `regexp` | `regexpi` |

Purpose

Compare first `n` characters of strings (case insensitive)

Syntax

```
TF = strncmpi(string,string,n)
TF = strncmpi(string,cellstr,n)
TF = strncmpi(cellstr,cellstr,n)
```

Description

`TF = strncmpi(string,string,n)` compares the first `n` characters of two strings for equality, ignoring any differences in letter case. The function returns a scalar logical 1 for equality, or scalar logical 0 for inequality.

`TF = strncmpi(string,cellstr,n)` compares the first `n` characters of a string with the first `n` characters of each element of a cell array of strings, ignoring letter case. The function returns a logical array the same size as the `cellstr` input in which logical 1 represents equality. The order of the input arguments is not important.

`TF = strncmpi(cellstr,cellstr,n)` compares each element of one cell array of strings with the same element of the other, ignoring letter case. `strncmpi` attempts to match only the first `n` characters of these strings. The function returns a logical array the same size as either input array.

Tips

- The `strncmpi` function is intended for comparison of character data. When used to compare numeric data, it returns logical `0`.
- Use `strncmp` for case-sensitive string comparisons.
- Any leading and trailing blanks in either of the strings are explicitly included in the comparison.
- The value returned by `strncmpi` is not the same as the C language convention.
- `strncmpi` supports international character sets.

Input Arguments

string

A single character string or n-by-1 array of strings.

cellstr

A cell array of strings.

n

Maximum number of characters to compare. Must be a scalar, integer-valued double.

Output Arguments

TF

When both inputs are character arrays, `TF` is a scalar logical value. This value is logical 1 (`true`) if the size and content of both arrays are equal, and logical 0 (`false`) if they are not.

When either or both inputs are a cell array of strings, `TF` is an array of logical ones and zeros. This array is the same size as the input cell array(s), and contains logical 1 (`true`) for those elements of the input arrays that are a match, and logical 0 (`false`) for those elements that are not.

Examples

From a list of 10 MATLAB functions, find those that apply to using a camera. Do the comparison without sensitivity to letter case:

```
function_list = {'calendar' 'case' 'camdolly' 'circshift' ...
                 'caxis' 'Camtarget' 'cast' 'camorbit' ...
                 'callib' 'cart2sph'};

strncmpi(function_list, 'CAM', 3)
ans =
     0     0     1     0     0     1     0     1     0     0

function_list{strncmpi(function_list, 'CAM', 3)}
ans =
   camdolly
ans =
   Camtarget
ans =
   camorbit
```

See Also

`strcmpi` | `strncmp` | `strcmp` | `strfind` | `regexpi` | `regexp` |

strread

Purpose

Read formatted data from string

Note `strread` is not recommended. Use `textscan` instead.

Syntax

```
A = strread('str')
[A, B, ...] = strread('str')
[A, B, ...] = strread('str', 'format')
[A, B, ...] = strread('str', 'format', N)
[A, B, ...] = strread('str', 'format', N, param, value, ...)
```

Description

`A = strread('str')` reads numeric data from input string `str` into a 1-by-`N` vector `A`, where `N` equals the number of whitespace-separated numbers in `str`. Use this form only with strings containing numeric data. See "Example 1" on page 1-5042 below.

`[A, B, ...]  = strread('str')` reads numeric data from the string input `str` into scalar output variables `A`, `B`, and so on. The number of output variables must equal the number of whitespace-separated numbers in `str`. Use this form only with strings containing numeric data. See "Example 2" on page 1-5043 below.

`[A, B, ...]  = strread('str', 'format')` reads data from `str` into variables `A`, `B`, and so on using the specified `format`. The number of output variables `A`, `B`, etc. must be equal to the number of format specifiers (e.g., `%s` or `%d`) in the `format` argument. You can read all of the data in `str` to a single output variable as long as you use only one format specifier in the command. See "Example 4" on page 1-5043 and "Example 5" on page 1-5044 below.

The table Formats for strread on page 1-5039 lists the valid format specifiers. More information on using formats is available under "Formats" on page 1-5042 in the "Tips" on page 1-5041 section below.

`[A, B, ...]  = strread('str', 'format', N)` reads data from `str` reusing the `format` string `N` times, where `N` is an integer greater than zero. If `N` is -1, `strread` reads the entire string. When `str` contains

only numeric data, you can set `format` to the empty string (`''`). See "Example 3" on page 1-5043 below.

`[A, B, ...]  = strread('str', 'format', N, param, value, ...)` customizes `strread` using `param/value` pairs, as listed in the table Parameters and Values for strread on page 1-5040 below. When `str` contains only numeric data, you can set `format` to the empty string (`''`). The `N` argument is optional and may be omitted entirely. See "Example 7" on page 1-5045 below.

Formats for strread

Format	Action	Output
Literals (ordinary characters)	Ignore the matching characters. For example, in a string that has `Dept` followed by a number (for department number), to skip the `Dept` and read only the number, use `'Dept'` in the `format` string.	None
`%d`	Read a signed integer value.	Double array
`%u`	Read an integer value.	Double array
`%f`	Read a floating-point value.	Double array
`%s`	Read a white-space separated string.	Cell array of strings
`%q`	Read a double quoted string, ignoring the quotes.	Cell array of strings
`%c`	Read characters, including white space.	Character array
`%[...]`	Read the longest string containing characters specified in the brackets.	Cell array of strings

Formats for strread (Continued)

Format	Action	Output
`%[^...]`	Read the longest nonempty string containing characters that are not specified in the brackets.	Cell array of strings
`%*...`	Ignore the characters following `*`. See “Example 8” on page 1-5045 below.	No output
`%w...`	Read field width specified by `w`. The `%f` format supports `%w.pf`, where `w` is the field width and `p` is the precision.	

Parameters and Values for strread

param	value	Action
`whitespace`	Any from the list below: `\b` Backspace `\n` New line `\r` Carriage return `\t` Horizontal tab `\\` Backslash `%%` Percent sign `''` Single quotation mark	Treats vector of characters, `*`, as white space. Default is `\b\r\n\t`.
`delimiter`	Delimiter character	Specifies delimiter character. Default is one or more whitespace characters.
`expchars`	Exponent characters	Default is `eEdD`.

Parameters and Values for strread (Continued)

param	value	Action
bufsize	Positive integer	Specifies the maximum string length, in bytes. Default is 4095.
commentstyle	matlab	Ignores characters after %.
commentstyle	shell	Ignores characters after #.
commentstyle	c	Ignores characters between /* and */.
commentstyle	c++	Ignores characters after //.
emptyvalue	Value to return for empty numeric fields in delimited files	Default is NaN.

Tips

If you terminate the input string with a newline character (\n), strread returns arrays of equal size by padding arrays of lesser size with the emptyvalue character:

```
[A,B,C] = strread(sprintf('5,7,1,9\n'),'%d%d%d', ...
              'delimiter', ',', 'emptyvalue',NaN)
A =
     5
     9
B =
     7
   NaN
C =
     1
   NaN
```

If you remove the \n from the input string of this example, array A continues to be a 2-by-1 array, but B and C are now 1-by-1.

Delimiters

If your data uses a character other than a space as a delimiter, you must use the strread parameter 'delimiter' to specify the delimiter. For example, if the string str used a semicolon as a delimiter, you would use this command:

```
[names, types, x, y, answer] = strread(str,'%s %s %f ...
    %d %s','delimiter',';')
```

Formats

The format string determines the number and types of return arguments. The number of return arguments must match the number of conversion specifiers in the format string.

The strread function continues reading str until the entire string is read. If there are fewer format specifiers than there are entities in str, strread reapplies the format specifiers, starting over at the beginning. See "Example 5" on page 1-5044 below.

The format string supports a subset of the conversion specifiers and conventions of the C language fscanf routine. White-space characters in the format string are ignored.

Preserving White-Space

If you want to preserve leading and trailing spaces in a string, use the whitespace parameter as shown here:

```
str = '   An  example      of preserving     spaces      ';

strread(str, '%s', 'whitespace', '')
ans =
    '   An  example      of preserving     spaces      '
```

Examples

Example 1

Read numeric data into a 1-by-5 vector:

```
a = strread('0.41 8.24 3.57 6.24 9.27')
a =
    0.4100    8.2400    3.5700    6.2400    9.2700
```

Example 2

Read numeric data into separate scalar variables:

```
[a b c d e] = strread('0.41 8.24 3.57 6.24 9.27')
a =
    0.4100
b =
    8.2400
c =
    3.5700
d =
    6.2400
e =
    9.2700
```

Example 3

Read the only first three numbers in the string, also formatting as floating point:

```
a = strread('0.41 8.24 3.57 6.24 9.27', '%4.2f', 3)

a =
    0.4100
    8.2400
    3.5700
```

Example 4

Truncate the data to one decimal digit by specifying format `%3.1f`. The second specifier, `%*1d`, tells `strread` not to read in the remaining decimal digit:

```
a = strread('0.41 8.24 3.57 6.24 9.27', '%3.1f %*1d')

a =
```

```
    0.4000
    8.2000
    3.5000
    6.2000
    9.2000
```

Example 5

Read six numbers into two variables, reusing the format specifiers:

```
[a b] = strread('0.41 8.24 3.57 6.24 9.27 3.29', '%f %f')

a =
    0.4100
    3.5700
    9.2700
b =
    8.2400
    6.2400
    3.2900
```

Example 6

Read string and numeric data to two output variables. Ignore commas in the input string:

```
str = 'Section 4, Page 7, Line 26';

[name value] = strread(str, '%s %d,')
name =
    'Section'
    'Page'
    'Line'
value =
     4
     7
    26
```

Example 7

Read the string used in the last example, but this time delimiting with commas instead of spaces:

```
str = 'Section 4, Page 7, Line 26';

[a b c] = strread(str, '%s %s %s', 'delimiter', ',')
a =
    'Section 4'
b =
    'Page 7'
c =
    'Line 26'
```

Example 8

Read selected portions of the input string:

```
str = '<table border=5 width="100%" cellspacing=0>';

[border width space] = strread(str, ...
   '%*s%*s %c %*s "%4s" %*s %c', 'delimiter', '= ')
border =
     5
width =
    '100%'
space =
     0
```

Example 9

Read the string into two vectors, restricting the `Answer` values to `T` and `F`. Also note that two delimiters (comma and space) are used here:

```
str = 'Answer_1: T, Answer_2: F, Answer_3: F';

[a b] = strread(str, '%s %[TF]', 'delimiter', ', ')
a =
    'Answer_1:'
```

```
    'Answer_2:'
    'Answer_3:'
b =
    'T'
    'F'
    'F'
```

See Also

`textscan` | `sscanf`

Purpose

Find and replace substring

Syntax

modifiedStr = strrep(*origStr*, *oldSubstr*, *newSubstr*)

Description

modifiedStr = strrep(*origStr*, *oldSubstr*, *newSubstr*) replaces all occurrences of the string *oldSubstr* within string *origStr* with the string *newSubstr*.

Tips

- strrep accepts input combinations of single strings, strings in scalar cells, and same-sized cell arrays of strings. If any inputs are cell arrays, strrep returns a cell array.
- The strrep function does not find empty strings for replacement. That is, when *origStr* and *oldSubstr* both contain the empty string (''), strrep does not replace '' with the contents of *newSubstr*.
- Before replacing strings, strrep finds all instances of *oldSubstr* in *origStr*, like the strfind function. For overlapping patterns, strrep performs multiple replacements. See the final example in the Examples section.

Examples

Replace text in a character array:

```
claim = 'This is a good example.';
new_claim = strrep(claim, 'good', 'great')
```

MATLAB returns:

```
new_claim =
This is a great example.
```

Replace text in a cell array:

```
c_files = {'c:\cookies.m'; ...
           'c:\candy.m';   ...
           'c:\calories.m'};
d_files = strrep(c_files, 'c:', 'd:')
```

MATLAB returns:

```
d_files =
    'd:\cookies.m'
    'd:\candy.m'
    'd:\calories.m'
```

Replace text in a cell array with values in a second cell array:

```
missing_info = {'Start: __'; ...
                'End: __'};

dates = {'01/01/2001'; ...
         '12/12/2002'};

complete = strrep(missing_info, '__', dates)
```

MATLAB returns:

```
complete =
    'Start: 01/01/2001'
    'End: 12/12/2002'
```

Compare the use of `strrep` and `regexprep` to replace a string with a repeated pattern:

```
repeats = 'abc 2 def 22 ghi 222 jkl 2222';
indices = strfind(repeats, '22')

using_strrep = strrep(repeats, '22', '*')
using_regexprep = regexprep(repeats, '22', '*')
```

MATLAB returns:

```
indices =
    11    18    19    26    27    28
```

```
using_strrep =
abc 2 def * ghi ** jkl ***

using_regexprep =
abc 2 def * ghi *2 jkl **
```

See Also

strfind | regexprep

strsplit

Purpose

Split string at specified delimiter

Syntax

```
C = strsplit(str)
C = strsplit(str,delimiter)
C = strsplit(str,delimiter,Name,Value)

[C,matches] = strsplit( ___ )
```

Description

`C = strsplit(str)` splits the string, `str`, at whitespace into the cell array of strings, `C`. A whitespace character is equivalent to any sequence in the set `{' ','\f','\n','\r','\t','\v'}`.

`C = strsplit(str,delimiter)` splits `str` at the delimiters specified by `delimiter`.

`C = strsplit(str,delimiter,Name,Value)` specifies additional delimiter options using one or more name-value pair arguments.

`[C,matches] = strsplit( ___ )` additionally returns a cell array of strings, `matches`, using any of the input arguments in the previous syntaxes. `matches` contains all occurrences of delimiters upon which `strsplit` splits `str`.

Input Arguments

str - Input text

string

Input text, specified as a string.

Data Types

`char`

delimiter - Delimiting characters

string | 1-by-n cell array of strings

Delimiting characters, specified as a single string or a 1-by-n cell array of strings. Strings specified in `delimiter` do not appear in the string fragments of the output `C`.

Specify multiple delimiters in a cell array of strings. Each element of the cell array must contain a single string in a single row. The `strsplit` function splits `str` on the elements of `delimiter` in the order in which they appear in the cell array.

`delimiter` can include the following escape sequences:

`\\`	Backslash
`\0`	Null
`\a`	Alarm
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab

Example: `','`

Example: `{'-',','}`

Data Types
`char | cell`

Name-Value Pair Arguments

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

Example: `'DelimiterType','RegularExpression'` instructs `strsplit` to treat `delimiter` as a regular expression.

'CollapseDelimiters' - Multiple delimiter handling

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

Multiple delimiter handling, specified as the comma-separated pair consisting of `'CollapseDelimiters'` and either `true` or `false`. If `true`, then consecutive delimiters in `str` are treated as one. If `false`, then consecutive delimiters are treated as separate delimiters, resulting in empty string `''` elements between matched delimiters.

Example: `'CollapseDelimiters',true`

'DelimiterType' - Delimiter type

`'Simple'` (default) | `'RegularExpression'`

Delimiter type, specified as the comma-separated pair consisting of `'DelimiterType'` and one of the following strings.

`'Simple'`	Except for escape sequences, `strsplit` treats `delimiter` as a literal string.
`'RegularExpression'`	`strsplit` treats `delimiter` as a regular expression.

In both cases, `delimiter` can include escape sequences.

Output Arguments

C - Parts of original string

cell array of strings

Parts of the original string, returned as a cell array of strings. `C` always contains one more element than `matches` contains. Consequently, if the original string, `str`, ends with a delimiter, then the last cell in `C` contains an empty string.

matches - Identified delimiters

cell array of strings

Identified delimiters, returned as a cell array of strings. `matches` always contains one fewer element than output `C` contains.

Examples

Split String on Whitespace

```
str = 'The rain in Spain.';
C = strsplit(str)

C =

    'The'    'rain'    'in'    'Spain.'
```

`C` is a cell array containing 4 strings.

Split String of Values on Specific Delimiter

Split a string of comma-separated values.

```
data = '1.21, 1.985, 1.955, 2.015, 1.885';
C = strsplit(data,', ')

C =

    '1.21'    '1.985'    '1.955'    '2.015'    '1.885'
```

Split a string of values, `data`, which contains the units `m/s` with an arbitrary number of white-space on either side of the text. The regular expression, `\s*`, matches any white-space character appearing zero or more times.

```
data = '1.21m/s1.985m/s 1.955 m/s2.015 m/s 1.885m/s';
[C,matches] = strsplit(data,'\s*m/s\s*','DelimiterType','RegularExpres

C =

    '1.21'    '1.985'    '1.955'    '2.015'    '1.885'    ''


matches =
```

```
    'm/s'    'm/s '    ' m/s'    ' m/s '    'm/s'
```

In this case, the last string in C is empty. This empty string is the string that follows the last matched delimiter.

Split Path String on File Separator

```
myPath = 'C:\work\matlab';
C = strsplit(myPath,'\')
```

```
C =

    'C:'    'work'    'matlab'
```

Split Text String with Multiple Delimiters

Split a string on `' '` and `'ain'`, treating multiple delimiters as one. Specify multiple delimiters in a cell array of strings.

```
str = 'The rain in Spain stays mainly in the plain.';
[C, matches] = strsplit(str,{' ', 'ain'},'CollapseDelimiters',true)
```

```
C =

    'The'    'r'    'in'    'Sp'    'stays'    'm'    'ly'    'in'    'th


matches =

    ' '    'ain '    ' '    'ain '    ' '    'ain'    ' '    ' '    ' '
```

Split the same string on whitespace and on `'ain'`, using regular expressions and treating multiple delimiters separately.

```
[C, matches] = strsplit(str, {'\s', 'ain'},'CollapseDelimiters',false, ..
                       'DelimiterType','RegularExpression')
```

```
C =
```

```
    'The'    'r'    ''    'in'    'Sp'    ''    'stays'    'm'    'ly

matches =

    ' '    'ain'    ' '    ' '    'ain'    ' '    ' '    'ain'    ' '
```

In this case, `strsplit` treats the two delimiters separately, so empty strings appear in output `C` between the consecutively matched delimiters.

Split Text with Multiple, Overlapping Delimiters

Split text on the strings `', '` and `', and '`.

```
str = 'bacon, lettuce, and tomato';
[C,matches] = strsplit(str,{', ',', and '})

C =

    'bacon'    'lettuce'    'and tomato'


matches =

    ', '    ', '
```

Because the command lists `', '` first and `', and '` contains `', '`, the `strsplit` function splits `str` on the first delimiter and never proceeds to the second delimiter.

If you reverse the order of delimiters, `', and '` takes priority.

```
str = 'bacon, lettuce, and tomato';
[C,matches] = strsplit(str,{', and ',', '})

C =

    'bacon'    'lettuce'    'tomato'
```

```
matches =

    ', '    ', and '
```

Split String Into Individual Characters

Split the string, `'abc'`, into individual characters using a regular expression.

The regular expression `'.'` represents any single character (including white space). You can specify this regular expression as a delimiter and extract single characters in the `matches` output. Use the name-value pair `'CollapseDelimiters',false` to treat consecutive characters as individual delimiters so that they appear as separate strings in `matches`. Use ~ in place of `C` to ignore the first output.

```
str = 'abc';
[~, matches] = strsplit(str,'.','DelimiterType', ...
            'RegularExpression','CollapseDelimiters',false)
```

```
matches =

    'a'    'b'    'c'
```

`strsplit` returns individual characters in the second output, `matches`.

See Also

`strjoin` | `strfind` | `regexp`

Concepts

- "Regular Expressions"

Purpose

Selected parts of string

Syntax

```
token = strtok(str)
token = strtok(str, delimiter)
[token, remain] = strtok('str', ...)
```

Description

`token = strtok(str)` parses input string `str` from left to right, returning part or all of that string in `token`. Using the white-space character as a delimiter, the `token` output begins at the start of `str`, skipping any delimiters that might appear at the start, and includes all characters up to either the next delimiter or the end of the string. White-space characters include space (ASCII 32), tab (ASCII 9), and carriage return (ASCII 13).

The `str` argument can be a string of characters enclosed in single quotation marks, a cell array of strings each enclosed in single quotation marks, or a variable representing either of the two. If `str` is a cell array of `N` strings, then `token` is a cell array of `N` tokens, with `token{1}` derived from `str{1}`, `token{2}` from `str{2}`, and so on.

`token = strtok(str, delimiter)` is the same as the above syntax except that you specify the delimiting character(s) yourself using the `delimiter` character vector input. White-space characters are not considered to be delimiters when using this syntax unless you include them in the `delimiter` argument. If the `delimiter` input specifies more than one character, MATLAB treats each character as a separate delimiter; it does not treat the multiple characters as a delimiting string. The number and order of characters in the `delimiter` argument is unimportant. Do not use escape sequences as delimiters. For example, use `char(9)` rather than `'\t'` for tab.

`[token, remain] = strtok('str', ...)` returns in `remain` that part of `str`, if any, that follows `token`. If no delimiters are found in the body of the input string, then the entire string (excluding any leading delimiting characters) is returned in `token`, and `remain` is an empty string (`''`). If `str` is a cell array of strings, `token` is a cell array of tokens and `remain` is a cell array of string remainders.

Examples

Example 1

This example uses the default white-space delimiter. Note that space characters at the start of the string are not included in the `token` output, but the space character that follows `token` is included in `remain`:

```
s = '  This is a simple example.';
[token, remain] = strtok(s)

token =
This
remain =
 is a simple example.
```

Example 2

Take a string of HTML code and break it down into segments delimited by the < and > characters. Write a `while` loop to parse the string and print each segment:

```
s = sprintf('%s%s%s%s', ...
'<ul class=continued><li class=continued>', ...
'<pre><a name="13474"></a>token = strtok', ...
'(''str'', delimiter)<a name="13475"></a>', ...
'token = strtok(''str'')');

remain = s;

while true
   [str, remain] = strtok(remain, '<>');
   if isempty(str),  break;  end
   disp(sprintf('%s', str))
end
```

Here is the output:

```
ul class=continued
li class=continued
pre
```

```
a name="13474"
/a
token = strtok('str', delimiter)
a name="13475"
/a
token = strtok('str')
```

Example 3

Using `strtok` on a cell array of strings returns a cell array of strings in `token` and a character array in `remain`:

```
s = {'all in good time'; ...
     'my dog has fleas'; ...
     'leave no stone unturned'};

remain = s;

for k = 1:4
   [token, remain] = strtok(remain);
   token
end
```

Here is the output:

```
token =
    'all'
    'my'
    'leave'
token =
    'in'
    'dog'
    'no'
token =
    'good'
    'has'
    'stone'
token =
```

```
'time'
'fleas'
'unturned'
```

See Also

strfind | strncmp | strcmp | textscan | strsplit

Purpose

Remove leading and trailing white space from string

Syntax

```
S = strtrim(str)
C = strtrim(cstr)
```

Description

`S = strtrim(str)` returns a copy of string `str` with all leading and trailing white-space characters removed. A white-space character is one for which the `isspace` function returns logical `1` (`true`).

`C = strtrim(cstr)` returns a copy of the cell array of strings `cstr` with all leading and trailing white-space characters removed from each string in the cell array.

Examples

Remove the leading white-space characters (spaces and tabs) from `str`:

```
str = sprintf('  \t   Remove    leading white-space')
str =
        Remove    leading white-space

str = strtrim(str)
str =
Remove    leading white-space
```

Remove leading and trailing white-space from the cell array of strings:

```
cstr = {'     Trim leading white-space';
        'Trim trailing white-space     '};

cstr = strtrim(cstr)
cstr =
    'Trim leading white-space'
    'Trim trailing white-space'
```

See Also

`isspace` | `cellstr` | `deblank` | `strjust`

Purpose

Create structure array

Syntax

```
s = struct
s = struct(field,value)
s = struct(field1,value1,...,fieldN,valueN)
s = struct([])

s = struct(obj)
```

Description

`s = struct` creates a scalar (1-by-1) structure with no fields.

`s = struct(field,value)` creates a structure array with the specified field and values.

- If `value` is *not* a cell array, then `s` is a scalar structure, where `s.(field) = value`.
- If `value` is a cell array, then `s` is a structure array with the same dimensions as `value`. Each element of `s` contains the corresponding element of `value`. For example, `s = struct('f',{'a','b'})` returns `s(1).f = 'a'` and `s(2).f = 'b'`.
- If `value` is an empty cell array `{}`, then `s` is an empty (0-by-0) structure.

`s = struct(field1,value1,...,fieldN,valueN)` creates a structure array with multiple fields. Any nonscalar cell arrays in the set `value1,...,valueN` must have the same dimensions.

- If none of the `value` inputs is a cell array, or all are scalar cell arrays, then `s` is a scalar structure.
- If any of the `value` inputs are nonscalar cell arrays, then `s` has the same dimensions as the nonscalar cell arrays. For any `value` that is a scalar cell array or an array of any other data type, `struct` inserts the contents of `value` in the relevant field for all elements of `s`.
- If any `value` input is an empty cell array, `{}`, then output `s` is an empty (0-by-0) structure.

`s = struct([])` creates an empty (0-by-0) structure with no fields.

`s = struct(obj)` creates a structure with field names and values that correspond to properties of `obj`. MATLAB does not convert `obj`, but rather creates `s` as a new structure. This structure does not retain the class information, so private, protected, and hidden properties become public fields in `s`. The `struct` function issues a warning when you use this syntax.

Input Arguments

field - Field name

string

Field name, specified as a string. Valid field names begin with a letter, and can contain letters, digits, and underscores. The maximum length of a field name is the value that the `namelengthmax` function returns.

value - Values within structure field

cell array | scalar | vector | multidimensional array

Values within structure field, specified as a cell array or as a scalar, vector, or multidimensional array of any other data type.

If none of the `value` inputs is a cell array, or all of the `value` inputs are scalar cell arrays, then output `s` is a scalar structure. Otherwise, `value` inputs that are nonscalar cell arrays must have the same dimensions, and output `s` also has those dimensions. For any `value` that is a scalar cell array or an array of any other data type, `struct` inserts the contents of `value` in the relevant field for all elements of `s`.

If any `value` input is an empty cell array, `{}`, then output `s` is an empty structure array.

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

obj - Object

any nonfundamental class

Object of any class, except the fundamental data types such as `double` or `char`.

Examples

Structure with One Field

Create a nonscalar structure with one field, `f`.

```
field = 'f';
value = {'some text';
         [10, 20, 30];
         magic(5)};
s = struct(field,value)

s =
3x1 struct array with fields:
    f
```

View the contents each element.

```
s.f

ans =
some text

ans =
    10    20    30

ans =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

When you access a field of a nonscalar structure, such as `s.f`, MATLAB returns a comma-separated list. In this case, `s.f` is equivalent to `s(1).f, s(2).f, s(3).f`.

Structure with Multiple Fields

Create a nonscalar structure with several fields.

```
field1 = 'f1';  value1 = zeros(1,10);
field2 = 'f2';  value2 = {'a', 'b'};
field3 = 'f3';  value3 = {pi, pi.^2};
field4 = 'f4';  value4 = {'fourth'};

s = struct(field1,value1,field2,value2,field3,value3,field4,value4)

s =
1x2 struct array with fields:
    f1
    f2
    f3
    f4
```

The cell arrays for `value2` and `value3` are 1-by-2, so `s` is also 1-by-2. Because `value1` is a numeric array and not a cell array, both `s(1).f1` and `s(2).f1` have the same contents. Similarly, because the cell array for `value4` has a single element, `s(1).f4` and `s(2).f4` have the same contents.

```
s(1)

ans =
    f1: [0 0 0 0 0 0 0 0 0 0]
    f2: 'a'
    f3: 3.1416
    f4: 'fourth'

s(2)

ans =
    f1: [0 0 0 0 0 0 0 0 0 0]
    f2: 'b'
    f3: 9.8696
    f4: 'fourth'
```

Fields that Contain Cell Arrays

Create a structure with a field that contains a cell array.

```
field = 'mycell';
value = {{'a','b','c'}};
s = struct(field,value)
```

```
s =
    mycell: {'a'  'b'  'c'}
```

Empty Structure Array

Create an empty structure with several fields.

```
s = struct('a',{},'b',{},'c',{})
```

```
s =
0x0 struct array with fields:
    a
    b
    c
```

Nested Structure Array

Create a nested structure: `a` contains field `b`, which in turn contains fields `c` and `d`.

```
a.b = struct('c',{},'d',{})
```

```
a =
    b: [0x0 struct]
```

View the names of the fields of nested structure `a.b`.

```
fieldnames(a.b)
```

```
ans =
    'c'
    'd'
```

See Also

`isfield` | `isstruct` | `fieldnames` | `orderfields` | `rmfield` | `substruct` | `cell2struct` | `struct2cell`

Related Examples

- "Create a Structure Array"
- "Access Data in a Structure Array"
- "Generate Field Names from Variables"

struct2cell

Purpose

Convert structure to cell array

Syntax

```
c = struct2cell(s)
```

Description

`c = struct2cell(s)` converts the `m`-by-`n` structure `s` (with `p` fields) into a `p`-by-`m`-by-`n` cell array `c`.

If structure `s` is multidimensional, cell array `c` has size `[p size(s)]`.

Examples

The commands

```
clear s, s.category = 'tree';
s.height = 37.4; s.name = 'birch';
```

create the structure

```
s =
    category: 'tree'
      height: 37.4000
        name: 'birch'
```

Converting the structure to a cell array,

```
c = struct2cell(s)

c =
    'tree'
    [37.4000]
    'birch'
```

See Also

`cell2struct` | `cell` | `iscell` | `struct` | `isstruct` | `fieldnames`

How To

- dynamic field names

Purpose

Apply function to each field of scalar structure

Syntax

```
[A1,...,An] = structfun(func,S)
[A1,...,An] = structfun(func,S,Name,Value)
```

Description

`[A1,...,An] = structfun(func,S)` applies the function specified by function handle `func` to each field of scalar structure `S`. Output arrays `A1,...,An`, where `n` is the number of outputs from function `func`, contain the outputs from the function calls.

`[A1,...,An] = structfun(func,S,Name,Value)` calls function `func` with additional options specified by one or more `Name,Value` pair arguments. Possible values for `Name` are `'UniformOutput'` or `'ErrorHandler'`.

Input Arguments

func

Handle to a function that accepts a single input argument and returns `n` output arguments.

If function `func` corresponds to more than one function file (that is, if `func` represents a set of overloaded functions), MATLAB determines which function to call based on the class of the input arguments.

S

Scalar structure.

Name-Value Pair Arguments

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

'UniformOutput'

Logical value, as follows:

`true (1)`	Indicates that for all inputs, each output from function `func` is a cell array or a scalar value that is always of the same type. The `structfun` function combines the outputs in arrays `A1,...,An`, where `n` is the number of function outputs. Each output array is of the same type as the individual function outputs.
`false (0)`	Requests that the `structfun` function combine the outputs into scalar structures `A1,...,An`, with the same fields as input structure `S`. The outputs of function `func` can be of any size or type.

Default: `true`

'ErrorHandler'

Handle to a function that catches any errors that occur when MATLAB attempts to execute function `func`. Define this function so that it rethrows the error or returns valid outputs for function `func`.

MATLAB calls the specified error-handling function with two input arguments:

- A structure with these fields:

`identifier`	Error identifier.
`message`	Error message text.
`index`	Linear index corresponding to the element of the input cell array at the time of the error.

- The set of input arguments to function `func` at the time of the error.

Output Arguments

A1,...,An

Arrays that collect the `n` outputs from function `func`.

If `UniformOutput` is `true` (the default):

- The individual outputs from function `func` must be scalar values (numeric, logical, character, or structure) or cell arrays.
- The class of a particular output argument must be the same for each input. The class of the corresponding output array is the same as the class of the individual outputs from function `func`.
- Each array `A` is a column vector whose length equals the number of fields in `S`. The `structfun` function applies function `func` to the fields of `S` in the same order as that returned by the `fieldnames` function.

If `UniformOutput` is `false`, each array `A` is a scalar structure with the same fields as input `S`.

Examples

Create a scalar structure, and count the number of characters in each field.

```
s.f1 = 'Sunday';
s.f2 = 'Monday';
s.f3 = 'Tuesday';
s.f4 = 'Wednesday';
s.f5 = 'Thursday';
s.f6 = 'Friday';
s.f7 = 'Saturday';

lengths = structfun(@numel, s)
```

Shorten the text in each field of `s`, created in the previous example. Because the output is nonscalar, set `UniformOutput` to `false`.

```
shortNames = structfun(@(x) ( x(1:3) ), s, 'UniformOutput', false)
```

The syntax `@(x)` creates an anonymous function.

Define and call a custom error handling function.

```
function result = errorfun(errorinfo, field)
```

```
   warning(errorinfo.identifier, errorinfo.message);
   result = NaN;
end

mystruct.f1 = 'text';
myresult = structfun(@(x) x^2, mystruct, 'ErrorHandler', @errorfun)
```

See Also

cellfun | arrayfun | function_handle | cell2mat | spfun

Tutorials

- "Anonymous Functions"

Purpose

Concatenate strings vertically

Note `strvcat` is not recommended. Use `char` instead. Unlike `strvcat`, the `char` function does not ignore empty strings.

Syntax

```
S = strvcat(t1, t2, t3, ...)
S = strvcat(c)
```

Description

`S = strvcat(t1, t2, t3, ...)` forms the character array `S` containing the text strings (or string matrices) `t1,t2,t3,...` as rows. Spaces are appended to each string as necessary to form a valid matrix. Empty arguments are ignored.

`S = strvcat(c)` when `c` is a cell array of strings, passes each element of `c` as an input to `strvcat`. Empty strings in the input are ignored.

Tips

If each text parameter, `ti`, is itself a character array, `strvcat` appends them vertically to create arbitrarily large string matrices.

Examples

The command `strvcat('Hello','Yes')` is the same as `['Hello';'Yes  ']`, except that `strvcat` performs the padding automatically.

```
t1 = 'first'; t2 = 'string'; t3 = 'matrix'; t4 = 'second';

S1 = strvcat(t1, t2, t3)        S2 = strvcat(t4, t2, t3)

S1 =                            S2 =

first                           second
string                          string
matrix                          matrix

S3 = strvcat(S1, S2)

S3 =
```

```
first
string
matrix
second
string
matrix
```

See Also

strcat | cat | vertcat | horzcat | int2str | mat2str | num2str | strings | special character

Purpose

Convert subscripts to linear indices

Syntax

```
linearInd = sub2ind(matrixSize, rowSub, colSub)
linearInd = sub2ind(arraySize, dim1Sub, dim2Sub, dim3Sub,
   ...)
```

Description

linearInd = sub2ind(*matrixSize*, *rowSub*, *colSub*) returns the linear index equivalents to the row and column subscripts *rowSub* and *colSub* for a matrix of size *matrixSize*. The *matrixSize* input is a 2-element vector that specifies the number of rows and columns in the matrix as [nRows, nCols]. The *rowSub* and *colSub* inputs are positive, whole number scalars or vectors that specify one or more row-column subscript pairs for the matrix. Example 3 demonstrates the use of vectors for the rowSub and colSub inputs.

linearInd = sub2ind(*arraySize*, *dim1Sub*, *dim2Sub*, *dim3Sub*, ...) returns the linear index equivalents to the specified subscripts for each dimension of an N-dimensional array of size *arraySize*. The *arraySize* input is an n-element vector that specifies the number of dimensions in the array. The *dimNSub* inputs are positive, whole number scalars or vectors that specify one or more row-column subscripts for the matrix.

The *rowSub* and *colSub* inputs must belong to the same class. The *linearInd* output is the same class as the subscript inputs.

If needed, sub2ind assumes that unspecified trailing subscripts are 1. See Example 2, below.

Examples

Example 1

This example converts the subscripts (2, 1, 2) for three-dimensional array A to a single linear index. Start by creating a 3-by-4-by-2 array A:

```
rng(0,'twister');   % Initialize random number generator.
A = rand(3, 4, 2)

A(:,:,1) =
    0.8147    0.9134    0.2785    0.9649
```

```
    0.9058    0.6324    0.5469    0.1576
    0.1270    0.0975    0.9575    0.9706
A(:,:,2) =
    0.9572    0.1419    0.7922    0.0357
    0.4854    0.4218    0.9595    0.8491
    0.8003    0.9157    0.6557    0.9340
```

Find the linear index corresponding to (2, 1, 2):

```
linearInd = sub2ind(size(A), 2, 1, 2)
linearInd =
    14
```

Make sure that these agree:

```
A(2, 1, 2)            A(14)
ans =                 and =
     0.4854               0.4854
```

Example 2

Using the 3-dimensional array `A` defined in the previous example, specify only 2 of the 3 subscript arguments in the call to `sub2ind`. The third subscript argument defaults to 1.

The command

```
linearInd = sub2ind(size(A), 2, 4)
ans =
    11
```

is the same as

```
linearInd = sub2ind(size(A), 2, 4, 1)
ans =
    11
```

Example 3

Using the same 3-dimensional input array A as in Example 1, accomplish the work of five separate sub2ind commands with just one.

Replace the following commands:

```
sub2ind(size(A), 3, 3, 2);
sub2ind(size(A), 2, 4, 1);
sub2ind(size(A), 3, 1, 2);
sub2ind(size(A), 1, 3, 2);
sub2ind(size(A), 2, 4, 1);
```

with a single command:

```
sub2ind(size(A), [3 2 3 1 2], [3 4 1 3 4], [2 1 2 2 1])
ans =
    21    11    15    19    11
```

Verify that these linear indices access the same array elements as their subscripted counterparts:

```
[A(3,3,2),   A(2,4,1),   A(3,1,2),   A(1,3,2),   A(2,4,1)]
ans =
    0.6557    0.1576    0.8003    0.7922    0.1576

A([21, 11, 15, 19, 11])
ans =
    0.6557    0.1576    0.8003    0.7922    0.1576
```

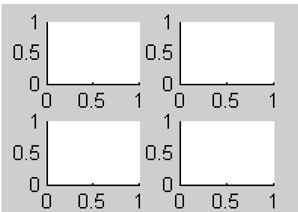
See Also

ind2sub | find | size

subplot

Purpose

Create axes in tiled positions



Syntax

```
subplot(m,n,p)
subplot(m,n,p,'replace')
subplot(m,n,p,'align')
subplot(m,n,P)
subplot(h)
subplot('Position',[left bottom width height])
subplot(...,prop1,value1,prop2,value2,...)
h = subplot(...)
```

Description

`subplot` divides the current figure into a grid that is numbered row-wise. Each block in the grid can contain an axes object which you can manipulate using `Axes Properties`. Subsequent plots are output to the current axes.

`subplot(m,n,p)` breaks the figure window into an `m`-by-`n` grid and creates an axes object in the `p`th location for the current plot, and returns the axes handle. MATLAB numbers the axes along the top row of the figure window, then the second row, etc. For example,

```
subplot(2,1,1), plot(x)
subplot(2,1,2), plot(y)
```

plots *x* on the top half of the window and *y* on the bottom half. The recently created axes becomes the current axes where plot command creates the graph.

`subplot(m,n,p,'replace')` deletes the specified axes object if it already exists and creates a new axes.

`subplot(m,n,p,'align')` places the axes so that the plot boxes are aligned, but does not prevent the labels and ticks from overlapping.

`subplot(m,n,P)`, where `P` is a vector, specifies an axes position that covers all the subplot positions listed in `P`, including those spanned by `P`. For example, `subplot(2,3,[2 5])` creates one axes spanning positions 2 and 5 only (because there are no intervening locations in the grid), while `subplot(2,3,[2 6])` creates one axes spanning positions 2, 3, 5, and 6.

`subplot(h)` makes the axes object with handle `h` the current axes for subsequent plotting commands.

`subplot('Position',[left bottom width height])` creates an axes at the position specified by a four-element vector. The `left`, `bottom`, `width`, and `height` values are normalized coordinates in the range from 0.0 to 1.0.

`subplot(...,prop1,value1,prop2,value2,...)` sets the specified property-value pairs on the subplot axes object. Available property/value pairs are described in `Axes Properties`. To add the subplot to a specific figure or uipanel, pass the handle as the value for the `jParent` property. You cannot specify both a `Parent` and a `Position`; that is, `subplot('Position',[left bottom width height], 'Parent',h)` is not a valid syntax.

`h = subplot(...)` returns the handle to the new `axes` object.

Tips

If a `subplot` specification causes a new axes object to overlap any existing axes, `subplot` deletes the existing axes object and uicontrol objects. However, if the `subplot` specification exactly matches the position of an existing axes object, the matching axes object is not deleted and it becomes the current axes.

`subplot(1,1,1)` or `clf` deletes all axes objects and returns to the default `subplot(1,1,1)` configuration.

You can omit the parentheses and specify `subplot` as

```
subplot mnp
```

where `m` refers to the row, `n` refers to the column, and `p` specifies the grid location.

Be aware when creating subplots from scripts that the `Position` property of subplots is not finalized until either

- A `drawnow` command is issued.
- MATLAB returns to await a user command.

That is, the value obtained for subplot `i` by the command

```
get(h(i),'position')
```

will not be correct until the script refreshes the plot or exits.

Special Case: subplot(111)

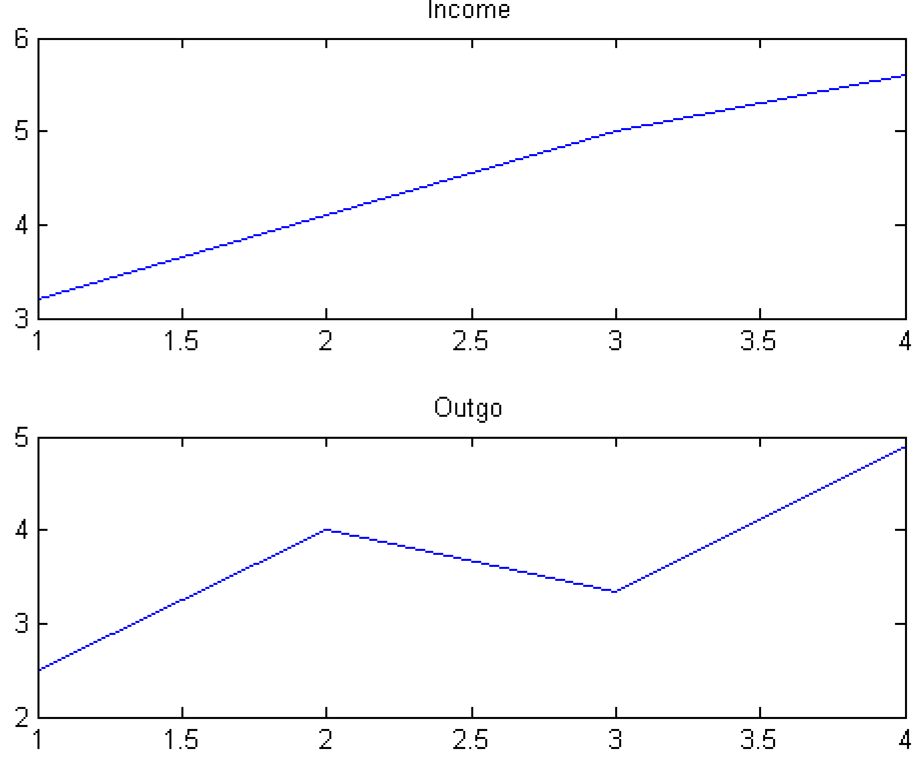
The command `subplot(111)` is not identical in behavior to `subplot(1,1,1)` and exists only for compatibility with previous releases. This syntax does not immediately create an axes object, but instead sets up the figure so that the next graphics command executes a `clf reset` (deleting all figure children) and creates a new axes object in the default position. This syntax does not return a handle, so it is an error to specify a return argument. (MATLAB implements this behavior by setting the figure's `NextPlot` property to `replace`.)

Examples

Upper and Lower Subplots with Titles

To plot `income` in the top half of a figure and `outgo` in the bottom half,

```
income = [3.2,4.1,5.0,5.6];
outgo = [2.5,4.0,3.35,4.9];
subplot(2,1,1); plot(income)
title('Income')
subplot(2,1,2); plot(outgo)
title('Outgo')
```



Subplots in Quadrants

The following illustration shows four subplot regions and indicates the command used to create each.

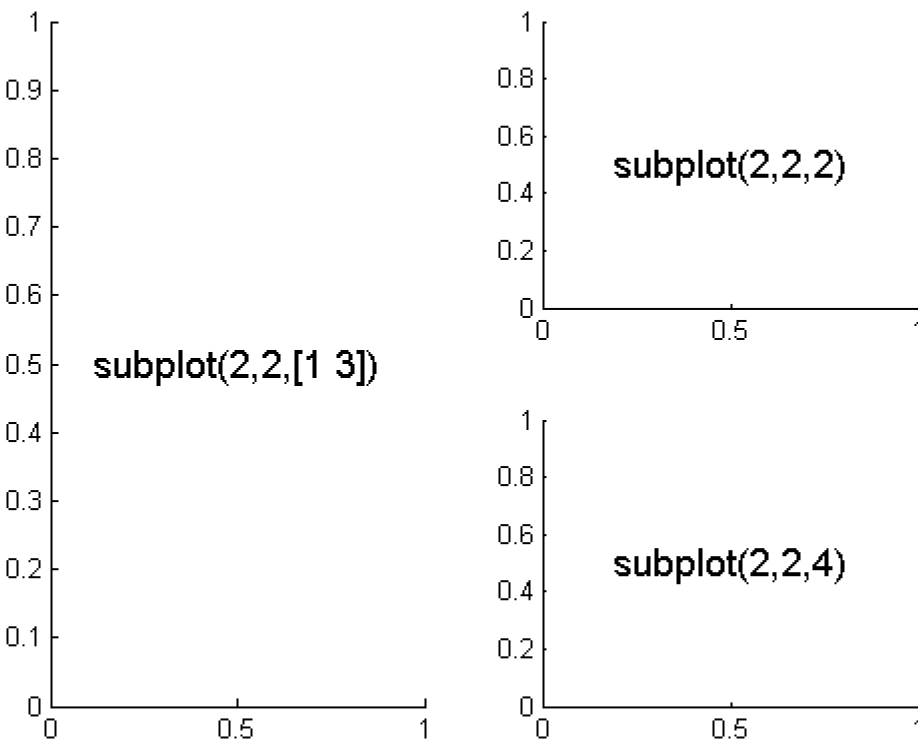
```
figure
subplot(2,2,1)
text(.5,.5,{'subplot(2,2,1)';'or subplot 221'},...
    'FontSize',14,'HorizontalAlignment','center')
subplot(2,2,2)
text(.5,.5,{'subplot(2,2,2)';'or subplot 222'},...
```

```
    'FontSize',14,'HorizontalAlignment','center')
subplot(2,2,3)
text(.5,.5,{'subplot(2,2,3)';'or subplot 223'},...
    'FontSize',14,'HorizontalAlignment','center')
subplot(2,2,4)
text(.5,.5,{'subplot(2,2,4)';'or subplot 224'},...
    'FontSize',14,'HorizontalAlignment','center')
```

Asymmetrical Subplots

The following combinations produce asymmetrical arrangements of subplots.

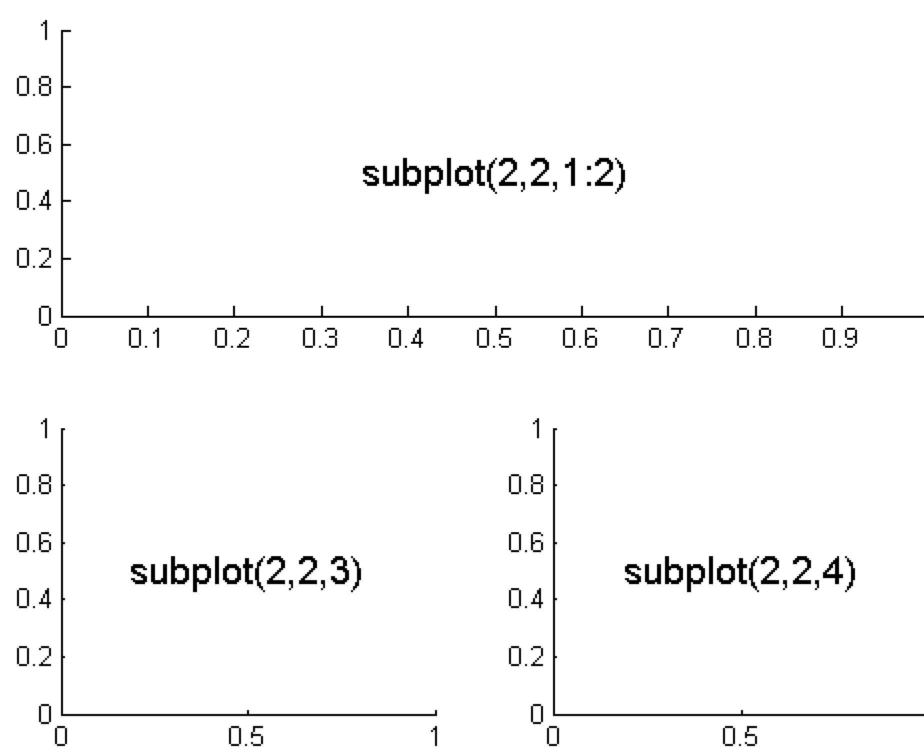
```
figure
subplot(2,2,[1 3])
text(.5,.5,'subplot(2,2,[1 3])',...
    'FontSize',14,'HorizontalAlignment','center')
subplot(2,2,2)
text(.5,.5,'subplot(2,2,2)',...
    'FontSize',14,'HorizontalAlignment','center')
subplot(2,2,4)
text(.5,.5,'subplot(2,2,4)',...
    'FontSize',14,'HorizontalAlignment','center')
```



You can also use the colon operator to specify multiple locations if they are in sequence.

```
figure
subplot(2,2,1:2)
text(.5,.5,'subplot(2,2,1:2)',...
    'FontSize',14,'HorizontalAlignment','center')
subplot(2,2,3)
text(.5,.5,'subplot(2,2,3)',...
    'FontSize',14,'HorizontalAlignment','center')
```

```
subplot(2,2,4)
text(.5,.5,'subplot(2,2,4)',...
    'FontSize',14,'HorizontalAlignment','center')
```



Suppressing Axis Ticks

When you create many subplots in a figure, the axes tickmarks, which are shown by default, can either be obscured or can cause axes to collapse.

- One way to get around this issue is to enlarge the figure to create enough space to properly display the tick labels.
- Another approach is to eliminate the clutter by suppressing xticks and yticks for subplots as data are plotted into them as,

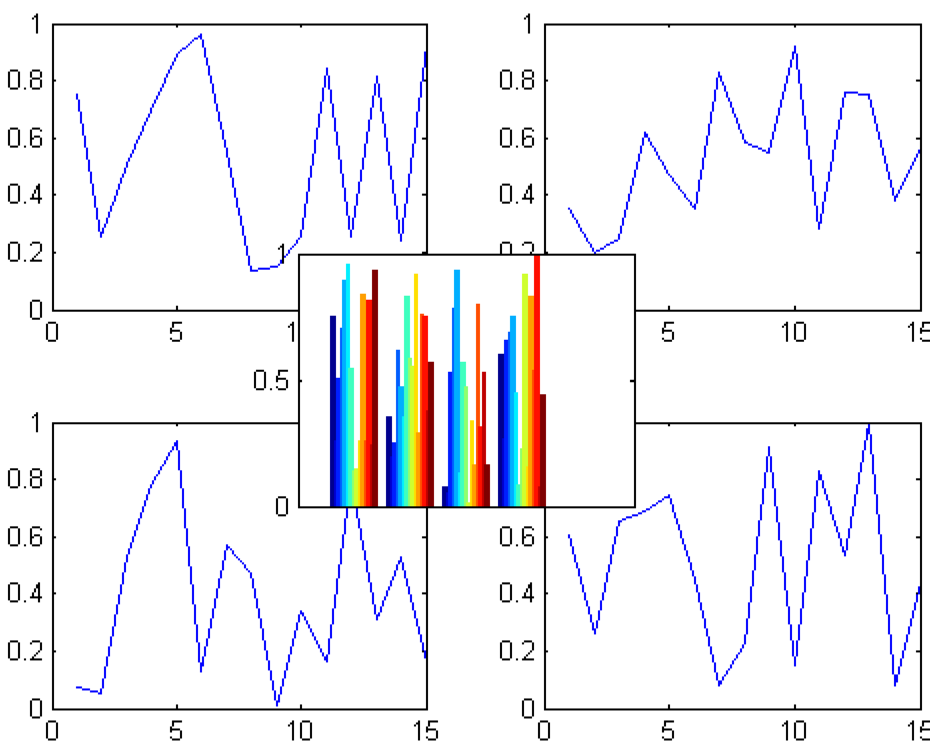
  ```
  h = subplot(10,1,1);
  set(h,'XTick',[],'YTick',[]);
  ```

  If the subplots are stacked, you can then display tick labels for the bottom axis.

Plotting Axes Over Subplots

Place a graph in the center, on top of four other graphs, using the `axes` and `subplot` functions. Data is random; your graphs can differ.

```
figure
y = zeros(4,15);
for k = 1:4
    y(k,:) = rand(1,15);
    subplot(2, 2, k)
    plot(y(k,:));
end
hax = axes('Position', [.35, .35, .3, .3]);
bar(hax,y,'EdgeColor','none')
set(hax,'XTick',[])
```



Alternatives

To add subplots to a figure, click one of the `New Subplot` icons in the Figure Palette, and slide right to select an arrangement of subplots. For details, see Plotting Tools — Interactive Plotting.

See Also

`axes` | `cla` | `clf` | `figure` | `gca`

How To

- "Displaying Multiple Plots per Figure"

subsasgn

Purpose

Subscripted assignment

Syntax

```
A = subsasgn(A, S, B)
```

Description

`A = subsasgn(A, S, B)` is called by MATLAB for the syntax `A(i) = B`, `A{i} = B`, or `A.i = B` when `A` is an object.

MATLAB uses the built-in `subsasgn` function to interpret indexed assignment statements. Modify the indexed assignment behavior of classes by overloading `subsasgn` in the class.

If `A` is a fundamental class (see "Fundamental MATLAB Classes"), then an indexed reference to `A` calls the built-in `subsasgn` function. It does not call a `subsasgn` method that you have overloaded for that class. Therefore, if `A` is an array of class `double`, and there is an `@double/subsasgn` method on your MATLAB path, the statement `A(I) = B` calls the MATLAB built-in `subsasgn` function.

Tips

Within a class's own methods, MATLAB calls the built-in `subsasgn`, not the class defined `subsasgn`. This behavior enables to use the default `subsasgn` behavior when defining specialized indexing for your class. See "subsref and subsasgn Within Class Methods — Built-In Called" for more information.

Input Arguments

A

Object

S

`struct` array with two fields, `type` and `subs`.

- `type` is a string containing `'()'`, `'{}'`, or `'.'`, where `'()'` specifies integer subscripts, `'{}'` specifies cell array subscripts, and `'.'` specifies subscripted structure fields.
- `subs` is a cell array or string containing the actual subscripts.

B

Assignment value (right-hand side)

Output Arguments

A

Result of evaluating assignment.

Examples

See how MATLAB calls `subsasgn` for the expression:

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

The syntax `A(1:2,:)  = B` calls `A = subsasgn(A,S,B)` where `S` is a 1-by-1 structure with `S.type = '()'` and `S.subs = {1:2,':'}`. The string `':'` indicates a colon used as a subscript.

See how MATLAB calls `subsasgn` for the expression:

```
A{1:2} = B;
```

The syntax `A{1:2} = B` calls `A = subsasgn(A,S,B)` where `S.type = '{}'` and `S.subs = {[1 2]}`.

See how MATLAB calls `subsasgn` for the expression:

```
A.field = B;
```

The syntax `A.field = B` calls `A = subsasgn(A,S,B)` where `S.type = '.'` and `S.subs = 'field'`.

See how MATLAB calls `subsasgn` for the expression:

```
A(1,2).name(3:5)=B;
```

Simple calls combine in a straightforward way for more complicated indexing expressions. In such cases, `length(S)` is the number of subscripting levels. For instance, `A(1,2).name(3:5)=B` calls `A=subsasgn(A,S,B)` where `S` is a 3-by-1 structure array with the following values:

`S(1).type = '()'`	`S(2).type = '.'`	`S(3).type = '()'`
`S(1).subs = {1,2}`	`S(2).subs = 'name'`	`S(3).subs = {[3 4 5]}`

Algorithms

In the assignment `A(J,K,...)  = B(M,N,...)`, subscripts `J`, `K`, `M`, `N`, and so on, can be scalar, vector, or arrays, when all the following are true:

- The number of subscripts specified for `B`, excluding trailing subscripts equal to 1, does not exceed the value returned by `ndims(B)`.
- The number of nonscalar subscripts specified for `A` equals the number of nonscalar subscripts specified for `B`. For example, `A(5,1:4,1,2) = B(5:8)` is valid because both sides of the equation use one nonscalar subscript.
- The order and length of all nonscalar subscripts specified for `A` matches the order and length of nonscalar subscripts specified for `B`. For example, `A(1:4, 3, 3:9) = B(5:8, 1:7)` is valid because both sides of the equation (ignoring the one scalar subscript `3`) use a 4-element subscript followed by a 7-element subscript.

  See `numel` for information concerning the use of `numel` with regards to the overloaded `subsasgn` function.

See Also

`subsref` | `substruct`

Tutorials

- “Indexed Reference and Assignment”

Purpose

Subscript indexing with object

Syntax

```
ind = subsindex(A)
```

Description

`ind = subsindex(A)` called by MATLAB for the expression `X(A)` when `A` is an object, unless such an expression results in a call to an overloaded `subsref` or `subsasgn` method for `X`. `subsindex` must return the value of the object as a zero-based integer index. (`ind` must contain integer values in the range `0` to `prod(size(X))-1`.) Call `subsindex` directly from an overloaded `subsref` or `subsasgn` method.

MATLAB invokes `subsindex` separately on all the subscripts in an expression, such as `X(A,B)`.

See Also

`subsasgn` | `subsasgn`

Tutorials

- "Using Objects as Indices"

subspace

Purpose

Angle between two subspaces

Syntax

```
theta = subspace(A,B)
```

Description

`theta = subspace(A,B)` finds the angle between two subspaces specified by the columns of `A` and `B`. If `A` and `B` are column vectors of unit length, this is the same as `acos(abs(A'*B))`.

Tips

If the angle between the two subspaces is small, the two spaces are nearly linearly dependent. In a physical experiment described by some observations `A`, and a second realization of the experiment described by `B`, `subspace(A,B)` gives a measure of the amount of new information afforded by the second experiment not associated with statistical errors of fluctuations.

Examples

Consider two subspaces of a Hadamard matrix, whose columns are orthogonal.

```
H = hadamard(8);
A = H(:,2:4);
B = H(:,5:8);
```

Note that matrices `A` and `B` are different sizes — `A` has three columns and `B` four. It is not necessary that two subspaces be the same size in order to find the angle between them. Geometrically, this is the angle between two hyperplanes embedded in a higher dimensional space.

```
theta = subspace(A,B)
theta =
    1.5708
```

That `A` and `B` are orthogonal is shown by the fact that `theta` is equal to $\pi/2$.

```
theta - pi/2
ans =
     0
```

Purpose

Redefine subscripted reference for objects

Syntax

```
B = subsref(A,S)
```

Description

`B = subsref(A,S)` is called by MATLAB for the syntax `A(i)`, `A{i}`, or `A.i` when `A` is an object. `S` is a `struct` array with two fields, `type` and `subs`.

The `type` field is string containing `'()'`, `'{}'`, or `'.'`, where `'()'` specifies integer subscripts, `'{}'` specifies cell array subscripts, and `'.'` specifies subscripted structure fields. The subs field is a cell array or a string containing the actual subscripts.

`B` is the result of the indexed expression.

MATLAB uses the built-in `subsref` function to interpret indexed references to objects. To modify the indexed reference behavior of objects, overload `subsref` in the class.

If `A` is a fundamental class (see "Fundamental MATLAB Classes"), then an indexed reference to `A` calls the built-in `subsref` function. It does not call a `subsref` method that you have overloaded for that class. Therefore, if `A` is an array of class `double`, and there is an `@double/subsref` method on your MATLAB path, the statement `A(I)` calls the MATLAB built-in `subsref` function.

Tips

Within a class's own methods, MATLAB calls the built-in `subsref`, not the class defined `subsref`. This behavior enables to use the default `subsref` behavior when defining specialized indexing for your class. See "subsref and subsasgn Within Class Methods — Built-In Called" for more information.

Examples

See how MATLAB calls `subsref` for the expression:

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

The syntax `A(1:2,:)` calls `B = subsref(A,S)` where `S` is a 1-by-1 structure with `S.type='()'` and `S.subs={1:2,':'}`. The string `':'` indicates a colon used as a subscript.

See how MATLAB calls `subsref` for the expression:

```
A{1:2}
```

The syntax `A{1:2}` calls `B = subsref(A,S)` where `S.type='{}'` and `S.subs={[1 2]}`.

See how MATLAB calls `subsref` for the expression:

```
A.field
```

The syntax `A.field` calls `B = subsref(A,S)` where `S.type='.'` and `S.subs='field'`.

See how MATLAB calls `subsref` for the expression:

```
A(1,2).name(3:5)
```

Simple calls combine in a straightforward way for more complicated indexing expressions. In such cases, `length(S)` is the number of subscript levels. For instance, `A(1,2).name(3:5)` calls `subsref(A,S)` where `S` is a 3-by-1 structure array with the following values:

`S(1).type='()'`	`S(2).type='.'`	`S(3).type='()'`
`S(1).subs={1,2}`	`S(2).subs='name'`	`S(3).subs={[3 4 5]}`

See Also

`numel` | `subsasgn` | `substruct`

Tutorials

- “Indexed Reference and Assignment”

Purpose

Create structure argument for subsasgn or subsref

Syntax

```
S = substruct(type1, subs1, type2, subs2, ...)
```

Description

S = substruct(type1, subs1, type2, subs2, ...) creates a structure with the fields required by an overloaded subsref or subsasgn method. Each type string must be one of '.', '()', or '{}'. The corresponding subs argument must be either a field name (for the '.' type) or a cell array containing the index vectors (for the '()' or '{}' types).

Output Arguments

S

struct with these fields:

- type: one of '.', '()', or '{}'
- subs: subscript values (field name or cell array of index vectors)

Examples

Call subsref with arguments equivalent to the syntax:

```
B = A(3,5).field;
```

where A is an object of a class that implements a subsref method

Use substruct to form the input struct, S:

```
S = substruct('()',{3,5},'.','field');
```

Call the class method:

```
B = subsref(A,S);
```

The struct created by substruct in this example contains:

```
S(1)

ans =
```

```
    type: '()'
    subs: {[3]  [5]}

S(2)

ans =

    type: '.'
    subs: 'field'
```

See Also

`subsasgn` | `subsref`

Tutorials

- "Indexed Reference and Assignment"

Purpose

Extract subset of volume data set

Syntax

```
[Nx,Ny,Nz,Nv] = subvolume(X,Y,Z,V,limits)
[Nx,Ny,Nz,Nv] = subvolume(V,limits)
Nv = subvolume(...)
```

Description

`[Nx,Ny,Nz,Nv] = subvolume(X,Y,Z,V,limits)` extracts a subset of the volume data set `V` using the specified axis-aligned `limits`. `limits = [xmin,xmax,ymin, ymax,zmin,zmax]` (Any `NaN`s in the limits indicate that the volume should not be cropped along that axis.)

The arrays `X`, `Y`, and `Z` define the coordinates for the volume `V`. The subvolume is returned in `NV` and the coordinates of the subvolume are given in `NX`, `NY`, and `NZ`.

`[Nx,Ny,Nz,Nv] = subvolume(V,limits)` assumes the arrays `X`, `Y`, and `Z` are defined as

```
[X,Y,Z] = meshgrid(1:N,1:M,1:P)
```

where `[M,N,P] = size(V)`.

`Nv = subvolume(...)` returns only the subvolume.

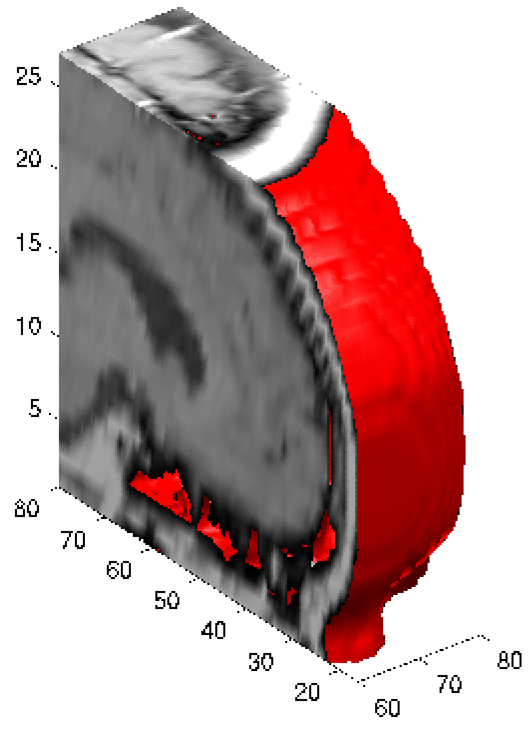
Examples

This example uses a data set that is a collection of MRI slices of a human skull. The data is processed in a variety of ways:

- The 4-D array is squeezed (`squeeze`) into three dimensions and then a subset of the data is extracted (`subvolume`).
- The outline of the skull is an isosurface generated as a patch (`p1`) whose vertex normals are recalculated to improve the appearance when lighting is applied (`patch`, `isosurface`, `isonormals`).
- A second patch (`p2`) with interpolated face color draws the end caps (`FaceColor`, `isocaps`).
- The view of the object is set (`view`, `axis`, `daspect`).
- A 100-element grayscale colormap provides coloring for the end caps (`colormap`).

- Adding lights to the right and left of the camera illuminates the object (`camlight`, `lighting`).

```
load mri
D = squeeze(D);
[x,y,z,D] = subvolume(D,[60,80,nan,80,nan,nan]);
p1 = patch(isosurface(x,y,z,D, 5),...
     'FaceColor','red','EdgeColor','none');
isonormals(x,y,z,D,p1);
p2 = patch(isocaps(x,y,z,D, 5),...
     'FaceColor','interp','EdgeColor','none');
view(3); axis tight; daspect([1,1,.4])
colormap(gray(100))
camlight right; camlight left; lighting gouraud
```



See Also

`isocaps` | `isonormals` | `isosurface` | `reducepatch` | `reducevolume` | `smooth3`

Purpose

Sum of array elements

Syntax

```
B = sum(A)
B = sum(A,dim)
B = sum(..., 'double')
B = sum(..., dim,'double')
B = sum(..., 'native')
B = sum(..., dim,'native')
```

Description

`B = sum(A)` returns sums along different dimensions of an array. If `A` is floating point, that is double or single, `B` is accumulated natively, that is in the same class as `A`, and `B` has the same class as `A`. If `A` is not floating point, `B` is accumulated in double and `B` has class double.

If `A` is a vector, `sum(A)` returns the sum of the elements.

If `A` is a matrix, `sum(A)` treats the columns of `A` as vectors, returning a row vector of the sums of each column.

If `A` is a multidimensional array, `sum(A)` treats the values along the first non-singleton dimension as vectors, returning an array of row vectors.

`B = sum(A,dim)` sums along the dimension of `A` specified by scalar `dim`. The `dim` input is an integer value from 1 to `N`, where `N` is the number of dimensions in `A`. Set `dim` to 1 to compute the sum of each column, 2 to sum rows, etc.

`B = sum(..., 'double')` and `B = sum(..., dim,'double')` perform additions in double-precision and return an answer of type `double`, even if `A` has data type `single` or an integer data type. This is the default for integer data types.

`B = sum(..., 'native')` and `B = sum(..., dim,'native')` perform additions in the native data type of `A` and return an answer of the same data type. This is the default for `single` and `double`.

If `A = int8(1:20)` then `sum(A)` accumulates in double and the result is `double(210)` while `sum(A,'native')` accumulates in `int8`, but overflows and saturates to `int8(127)`.

Tips

`sum(diag(A))` is the `trace` of `A`.

Examples

The magic square of order 3 is:

```
M = magic(3)
M =
     8     1     6
     3     5     7
     4     9     2
```

This is called a magic square because the sums of the elements in each column are the same.

```
sum(M)

ans =

    15    15    15
```

The sums of the elements in each row are also the same. You can sum the elements in each row by transposing `M` or by using the `dim` argument.

- Transposing `M`:

  ```
  sum(M')

  ans =

      15    15    15
  ```

- Using the `dim` argument:

  ```
  sum(M,2)

  ans =

      15
      15
      15
  ```

See Also

`cumsum` | `diff` | `isfloat` | `prod`

Purpose

Superclass names

Syntax

```
superclasses('ClassName')
superclasses(obj)
s = superclasses(...)
```

Description

`superclasses('ClassName')` displays the names of all visible superclasses of the MATLAB class with the name *ClassName*. Visible classes have a `Hidden` attribute value of `false` (the default).

`superclasses(obj)` `obj` is an instance of a MATLAB class. `obj` can be either a scalar object or an array of objects.

`s = superclasses(...)` returns the superclass names in a cell array of strings.

Examples

Get the name of the `hgsetget` class superclass:

```
superclasses('hgsetget')

Superclasses for class hgsetget:

    handle
```

See Also

`properties` | `methods` | `events` | `classdef`

Tutorials

- "Hierarchies of Classes — Concepts"

superiorto

Purpose

Establish superior class relationship

Syntax

```
superiorto('class1', 'class2', ...)
```

Description

`superiorto('class1', 'class2', ...)` establishes that the class invoking this function in its constructor has higher precedence than the classes in the argument list.

The `superiorto` function establishes a precedence that determines which object method MATLAB calls. Use this function only from a constructor that calls the `class` function to create an object. For classes defined with `classdef` statements, see "Class Precedence".

Examples

Show function dispatching:

`a` is an object of class `class_a`, `b` is an object of class `class_b`, and `c` is an object of class `class_c`. The constructor method for `class_c` contains the statement `superiorto('class_a')`. Then, either of the following two statements:

```
e = fun(a,c);
e = fun(c,a);
```

invokes `class_c/fun`.

If you call a function with two objects having an unspecified relationship, MATLAB considers the two objects to have equal precedence. In this case, MATLAB calls the left-most object method. So `fun(b,c)` calls `class_b/fun`, while `fun(c,b)` calls `class_c/fun`.

See Also

`inferiorto`

Purpose

Open MathWorks Technical Support Web page

Note `support` will be removed in a future release.

Syntax

```
support
```

Description

`support` opens the MathWorks Technical Support Web page, `http://www.mathworks.com/support`, in a Web browser.

This Web page contains resources including

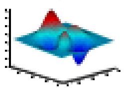
- A search engine, including an option for solutions to common problems
- Information about installation and licensing
- A patch archive for bug fixes you can download
- Other useful resources

See Also

`doc` | `web`

Purpose

3-D shaded surface plot



Syntax

```
surf(Z)
surf(Z,C)
surf(X,Y,Z)
surf(X,Y,Z,C)
surf(...,'PropertyName',PropertyValue)
surf(axes_handles,...)
h = surf(...)
```

Description

`surf(Z)` creates a three-dimensional shaded surface from the *z* components in matrix `Z`, using `x = 1:n` and `y = 1:m`, where `[m,n] = size(Z)`. The height, `Z`, is a single-valued function defined over a geometrically rectangular grid. `Z` specifies the color data, as well as surface height, so color is proportional to surface height.

`surf(Z,C)` plots the height of `Z`, a single-valued function defined over a geometrically rectangular grid, and uses matrix `C`, assumed to be the same size as `Z`, to color the surface.

`surf(X,Y,Z)` uses `Z` for the color data and surface height. `X` and `Y` are vectors or matrices defining the `x` and `y` components of a surface. If `X` and `Y` are vectors, `length(X)` = `n` and `length(Y)` = `m`, where `[m,n] = size(Z)`. In this case, the vertices of the surface faces are *(X(j), Y(i), Z(i,j))* triples. To create X and Y matrices for arbitrary domains, use the `meshgrid` function.

`surf(X,Y,Z,C)` uses `C` to define color. MATLAB performs a linear transformation on this data to obtain colors from the current colormap.

`surf(...,'PropertyName',PropertyValue)` specifies surface Surfaceplot along with the data.

`surf(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = surf(...)` returns a handle to a Surfaceplot graphics object.

Tips

`surf` does not accept complex inputs.

Algorithms

Consider a parametric surface parameterized by two independent variables, `i` and `j`, which vary continuously over a rectangle; for example, `1` ≤ `i` ≤ `m` and `1` ≤ `j` ≤ `n`. The three functions `x(i,j)`, `y(i,j)`, and `z(i,j)` specify the surface. When `i` and `j` are integer values, they define a rectangular grid with integer grid points. The functions `x(i,j)`, `y(i,j)`, and `z(i,j)` become three `m`-by-`n` matrices, `X`, `Y`, and `Z`. Surface color is a fourth function, `c(i,j)`, denoted by matrix `C`.

Each point in the rectangular grid can be thought of as connected to its four nearest neighbors.

```
       i-1,j
         |
i,j-1 - i,j - i,j+1
         |
       i+1,j
```

This underlying rectangular grid induces four-sided patches on the surface. To express this another way, `[X(:) Y(:) Z(:)]` returns a list of triples specifying points in 3-D space. Each interior point is connected to the four neighbors inherited from the matrix indexing. Points on the edge of the surface have three neighbors. The four points at the corners of the grid have only two neighbors. This defines a mesh of quadrilaterals or a *quad-mesh*.

You can specify surface color in two different ways: at the vertices or at the centers of each patch. In this general setting, the surface need not be a single-valued function of `x` and `y`. Moreover, the four-sided surface patches need not be planar. For example, you can have surfaces defined in polar, cylindrical, and spherical coordinate systems.

The `shading` function sets the shading. If the shading is `interp`, `C` must be the same size as `X`, `Y`, and `Z`; it specifies the colors at the vertices. The color within a surface patch is a bilinear function of the local

coordinates. If the shading is `faceted` (the default) or `flat`, `C(i,j)` specifies the constant color in the surface patch:

```
 (i,j)   -   (i,j+1)
   |   C(i,j)  |
(i+1,j)  -  (i+1,j+1)
```

In this case, `C` can be the same size as `X`, `Y`, and `Z` and its last row and column are ignored. Alternatively, its row and column dimensions can be one less than those of `X`, `Y`, and `Z`.

The `surf` function specifies the viewpoint using `view(3)`.

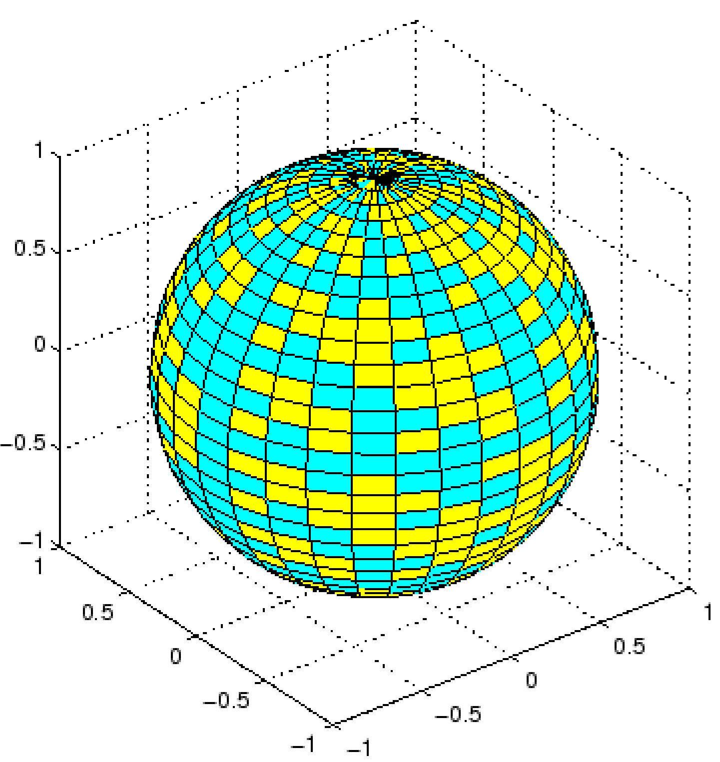
The range of `X`, `Y`, and `Z` or the current setting of `XLimMode`, `YLimMode`, and `ZLimMode` axes properties determines the axis labels. You can also set these properties using `axis` function.

The range of `C` or the current setting of the `CLim` and `CLimMode` axes properties determines the color scaling. You can also set the properties using `caxis` function. The scaled color values are indices into the current colormap.

Examples

Color a sphere with the pattern of +1s and -1s in a Hadamard matrix.

```
k = 5;
n = 2^k-1;
[x,y,z] = sphere(n);
c = hadamard(2^k);
surf(x,y,z,c);
colormap([1  1  0; 0  1  1])
axis equal
```



See Also

`surfc` | `axis` | `colormap` | `griddata` | `imagesc` | `mesh` | `meshgrid` | `pcolor` | `shading` | `trisurf` | `scatteredInterpolant` | `view` | Surfaceplot Properties

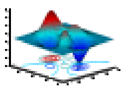
How To

- "Representing Data as a Surface"
- "Surface Plots of Nonuniformly Sampled Data"
- Coloring Mesh and Surface Plots

surfc

Purpose

Contour plot under a 3-D shaded surface plot



Syntax

```
surfc(Z)
surfc(Z,C)
surfc(X,Y,Z)
surfc(X,Y,Z,C)
surfc(...,'PropertyName',PropertyValue)
surfc(axes_handles,...)
h = surfc(...)
```

Description

`surfc(Z)` creates a contour plot under the three-dimensional shaded surface from the *z* components in matrix `Z`, using `x = 1:n` and `y = 1:m`, where `[m,n] = size(Z)`. The height, `Z`, is a single-valued function defined over a geometrically rectangular grid. `Z` specifies the color data, as well as surface height, so color is proportional to surface height.

`surfc(Z,C)` plots the height of `Z`, a single-valued function defined over a geometrically rectangular grid, and uses matrix `C`, assumed to be the same size as `Z`, to color the surface.

`surfc(X,Y,Z)` uses `Z` for the color data and surface height. `X` and `Y` are vectors or matrices defining the `x` and `y` components of a surface. If `X` and `Y` are vectors, `length(X) = n` and `length(Y) = m`, where `[m,n] = size(Z)`. In this case, the vertices of the surface faces are *(X(j), Y(i), Z(i,j))* triples. To create X and Y matrices for arbitrary domains, use the `meshgrid` function.

`surfc(X,Y,Z,C)` uses `C` to define color. MATLAB performs a linear transformation on this data to obtain colors from the current colormap.

`surfc(...,'PropertyName',PropertyValue)` specifies surface Surfaceplot along with the data.

`surfc(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = surfc(...)` returns a handle to a Surfaceplot graphics object.

Tips

`surfc` does not accept complex inputs.

Algorithms

Consider a parametric surface parameterized by two independent variables, `i` and `j`, which vary continuously over a rectangle; for example, `1` ≤ `i` ≤ `m` and `1` ≤ `j` ≤ `n`. The three functions `x(i,j)`, `y(i,j)`, and `z(i,j)` specify the surface. When `i` and `j` are integer values, they define a rectangular grid with integer grid points. The functions `x(i,j)`, `y(i,j)`, and `z(i,j)` become three `m`-by-`n` matrices, `X`, `Y`, and `Z`. Surface color is a fourth function, `c(i,j)`, denoted by matrix `C`.

Each point in the rectangular grid can be thought of as connected to its four nearest neighbors.

```
       i-1,j
         |
i,j-1 - i,j - i,j+1
         |
       i+1,j
```

This underlying rectangular grid induces four-sided patches on the surface. To express this another way, `[X(:) Y(:) Z(:)]` returns a list of triples specifying points in 3-D space. Each interior point is connected to the four neighbors inherited from the matrix indexing. Points on the edge of the surface have three neighbors. The four points at the corners of the grid have only two neighbors. This defines a mesh of quadrilaterals or a *quad-mesh*.

You can specify surface color in two different ways: at the vertices or at the centers of each patch. In this general setting, the surface need not be a single-valued function of `x` and `y`. Moreover, the four-sided surface patches need not be planar. For example, you can have surfaces defined in polar, cylindrical, and spherical coordinate systems.

The `shading` function sets the shading. If the shading is `interp`, `C` must be the same size as `X`, `Y`, and `Z`; it specifies the colors at the vertices. The color within a surface patch is a bilinear function of the local

coordinates. If the shading is `faceted` (the default) or `flat`, `C(i,j)` specifies the constant color in the surface patch:

```
 (i,j)   -   (i,j+1)
   |   C(i,j)  |
(i+1,j)  -  (i+1,j+1)
```

In this case, `C` can be the same size as `X`, `Y`, and `Z` and its last row and column are ignored. Alternatively, its row and column dimensions can be one less than those of `X`, `Y`, and `Z`.

The `surfc` function specifies the viewpoint using `view(3)`.

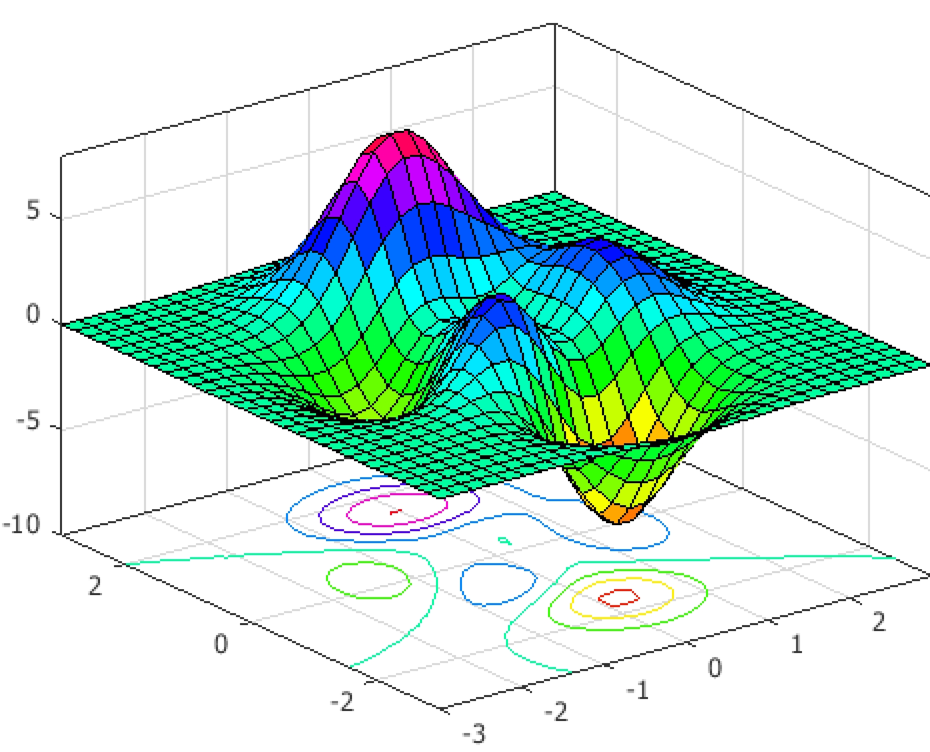
The range of `X`, `Y`, and `Z` or the current setting of the axes `XLimMode`, `YLimMode`, and `ZLimMode` properties (also set by the `axis` function) determines the axis labels.

The range of `C` or the current setting of the axes `CLim` and `CLimMode` properties (also set by the `caxis` function) determines the color scaling. The scaled color values are used as indices into the current colormap.

Examples

Display a surface plot and a contour plot of the `peaks` surface.

```
[X,Y,Z] = peaks(30);
surfc(X,Y,Z)
colormap hsv
```



See Also

surf | axis | caxis | colormap | contour | delaunay | imagesc | mesh | meshgrid | pcolor | shading | trisurf | view | Surfaceplot Properties

How To

- Representing a Matrix as a Surface
- Coloring Mesh and Surface Plots

surf2patch

Purpose

Convert surface data to patch data

Syntax

```
fvc = surf2patch(Z)
fvc = surf2patch(Z,C)
fvc = surf2patch(X,Y,Z)
fvc = surf2patch(X,Y,Z,C)
fvc = surf2patch(...,'triangles')
[f,v,c] = surf2patch(...)
```

Description

`fvc = surf2patch(h)`

converts the geometry and color data from the `surface` object identified by the handle `h` into patch format and returns the face, vertex, and color data in the struct `fvc`. You can pass this struct directly to the `patch` command.

`fvc = surf2patch(Z)` calculates the patch data from the surface's `ZData` matrix `Z`.

`fvc = surf2patch(Z,C)` calculates the patch data from the surface's `ZData` and `CData` matrices `Z` and `C`.

`fvc = surf2patch(X,Y,Z)` calculates the patch data from the surface's `XData`, `YData`, and `ZData` matrices `X`, `Y`, and `Z`.

`fvc = surf2patch(X,Y,Z,C)` calculates the patch data from the surface's `XData`, `YData`, `ZData`, and `CData` matrices `X`, `Y`, `Z`, and `C`.

`fvc = surf2patch(...,'triangles')` creates triangular faces instead of the quadrilaterals that compose surfaces.

`[f,v,c] = surf2patch(...)` returns the face, vertex, and color data in the three arrays `f`, `v`, and `c` instead of a struct.

Examples

The first example uses the `sphere` command to generate the `XData`, `YData`, and `ZData` of a surface, which is then converted to a patch. Note that the `ZData` (`z`) is passed to `surf2patch` as both the third and fourth arguments — the third argument is the `ZData` and the fourth argument is taken as the `CData`. This is because the `patch` command does not

automatically use the *z*-coordinate data for the color data, as does the `surface` command.

Also, because `patch` is a low-level command, you must set the `view` to 3-D and `shading` to `faceted` to produce the same results produced by the `surf` command.

```
[x y z] = sphere;
patch(surf2patch(x,y,z,z));
shading faceted; view(3)
```

In the second example `surf2patch` calculates face, vertex, and color data from a surface whose handle has been passed as an argument.

```
s = surf(peaks);
pause
patch(surf2patch(s));
delete(s)
shading faceted; view(3)
```

See Also

`patch` | `reducepatch` | `shrinkfaces` | `surface` | `surf`

surface

Purpose

Create surface object

Syntax

```
surface(Z)
surface(Z,C)
surface(X,Y,Z)
surface(X,Y,Z,C)
surface(x,y,Z)
surface(...'PropertyName',PropertyValue,...)
h = surface(...)
```

Properties

For a list of properties, see `Surface Properties`.

Description

`surface` is the low-level function for creating surface graphics objects. Surfaces are plots of matrix data created using the row and column indices of each element as the *x*- and *y*-coordinates and the value of each element as the *z*-coordinate.

`surface(Z)` plots the surface specified by the matrix `Z`. Here, `Z` is a single-valued function, defined over a geometrically rectangular grid.

`surface(Z,C)` plots the surface specified by `Z` and colors it according to the data in `C` (see "Examples").

`surface(X,Y,Z)` uses `C = Z`, so color is proportional to surface height above the *x*-*y* plane.

`surface(X,Y,Z,C)` plots the parametric surface specified by `X`, `Y`, and `Z`, with color specified by `C`.

`surface(x,y,Z)`, surface(x,y,Z,C) replaces the first two matrix arguments with vectors and must have `length(x) = n` and `length(y) = m` where `[m,n] = size(Z)`. In this case, the vertices of the surface facets are the triples `(x(j),y(i),Z(i,j))`. Note that `x` corresponds to the columns of `Z` and `y` corresponds to the rows of `Z`. For a complete discussion of parametric surfaces, see the `surf` function.

`surface(...'PropertyName',PropertyValue,...)` follows the `X`, `Y`, `Z`, and `C` arguments with property name/property value pairs to specify

additional surface properties. For a description of the properties, see `Surface Properties`.

`h = surface(...)` returns a handle to the created surface object.

Tips

`surface` does not respect the settings of the figure and axes `NextPlot` properties. It simply adds the surface object to the current axes.

If you do not specify separate color data (`C`), MATLAB uses the matrix (`Z`) to determine the coloring of the surface. In this case, color is proportional to values of `Z`. You can specify a separate matrix to color the surface independently of the data defining the area of the surface.

You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see `set` and `get` for examples of how to specify these data types).

`surface` provides convenience forms that allow you to omit the property name for the `XData`, `YData`, `ZData`, and `CData` properties. For example,

```
surface('XData',X,'YData',Y,'ZData',Z,'CData',C)
```

is equivalent to

```
surface(X,Y,Z,C)
```

When you specify only a single matrix input argument,

```
surface(Z)
```

MATLAB assigns the data properties as if you specified

```
surface('XData',[1:size(Z,2)],...
   'YData',[1:size(Z,1)],...
   'ZData',Z,...
   'CData',Z)
```

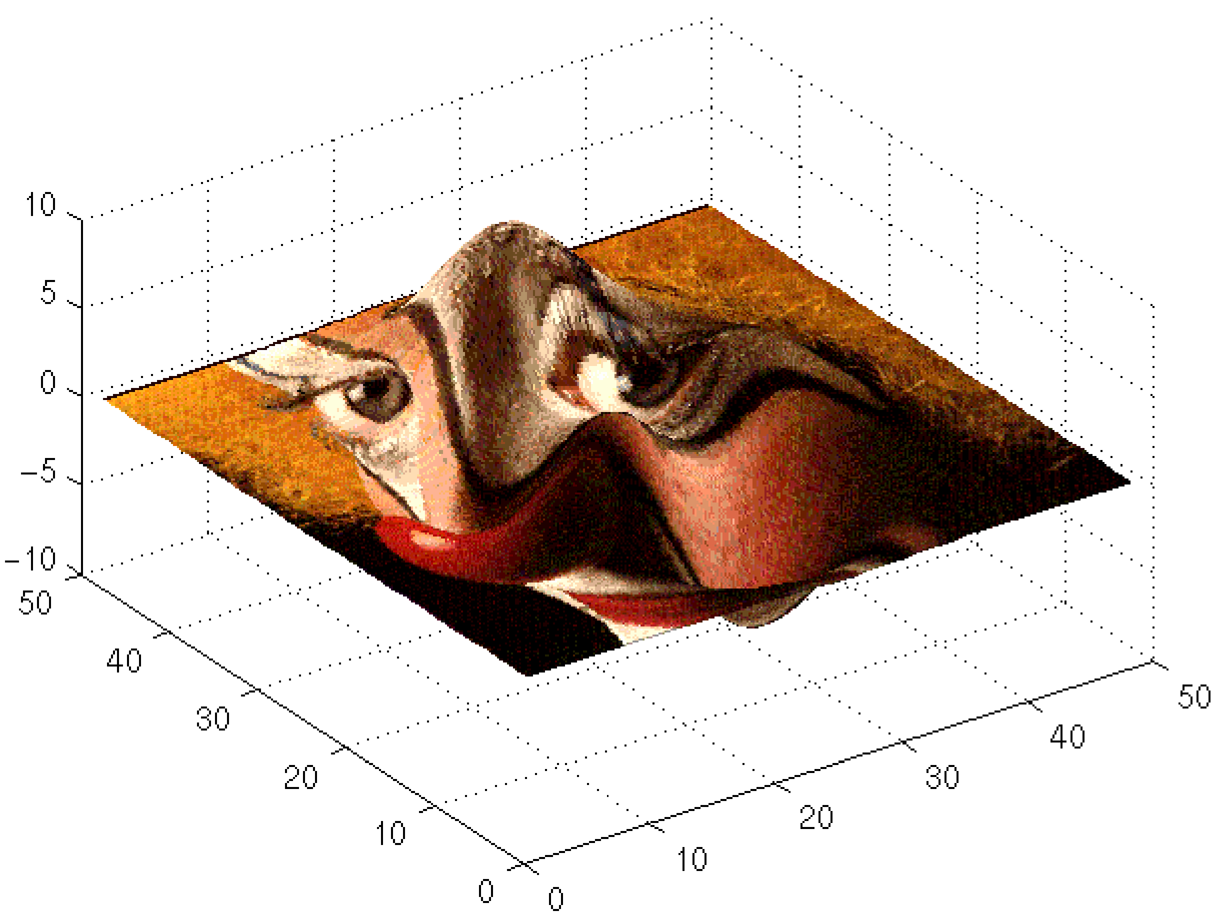
The `axis`, `caxis`, `colormap`, `hold`, `shading`, and `view` commands set graphics properties that affect surfaces. You can also set and query

surface property values after creating them using the `set` and `get` commands.

Examples

This example creates a surface using `peaks` to generate the data, and colors it using the clown image. The `ZData` is a 49-by-49 element matrix, while the `CData` is a 200-by-320 matrix. You must set the surface's `FaceColor` to `texturemap` to use `ZData` and `CData` of different dimensions.

```
load clown
surface(peaks,flipud(X),...
   'FaceColor','texturemap',...
   'EdgeColor','none',...
   'CDataMapping','direct')
colormap(map)
view(-35,45)
```



Note the use of the `surface(Z,C)` convenience form combined with property name/property value pairs.

Since the clown data (`X`) is typically viewed with the `image` command, which MATLAB normally displays with `'ij'` axis numbering and `direct CDataMapping`, this example reverses the data in the vertical direction using `flipud` and sets the `CDataMapping` property to `direct`.

Setting Default Properties

You can set default surface properties on the axes, figure, and `root object` levels:

```
set(0,'DefaultSurfaceProperty',PropertyValue...)
set(gcf,'DefaultSurfaceProperty',PropertyValue...)
set(gca,'DefaultSurfaceProperty',PropertyValue...)
```

where *`Property`* is the name of the surface property whose default value you want to set and `PropertyValue` is the value you are specifying. Use `set` and `get` to access the surface properties.

Tutorials

For examples, see Representing a Matrix as a Surface.

See Also

`ColorSpec` | `patch` | `pcolor` | `surf` | `Surface Properties`

Purpose

Surface properties

Creating Surface Objects

Use `surface` to create surface objects.

Modifying Properties

You can set and query graphics object properties in two ways:

- Property Editor is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see Setting Default Property Values.

See "Core Graphics Objects" for general information about this type of object.

Surface Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`AlphaData`

`m`-by-`n` matrix of `double` or `uint8`

Transparency data. A matrix of non-NaN values specifying the transparency of each face or vertex of the object. The `AlphaData` can be of class `double` or `uint8`.

MATLAB software determines the transparency in one of the following ways:

- Using the elements of `AlphaData` as transparency values (`AlphaDataMapping` set to `none`)
- Using the elements of `AlphaData` as indices into the current alphamap (`AlphaDataMapping` set to `direct`)

- Scaling the elements of `AlphaData` to range between the minimum and maximum values of the axes `ALim` property (`AlphaDataMapping` set to `scaled`, the default)

`AlphaDataMapping`
`none | direct | {scaled}`

Transparency mapping method. Determines how MATLAB interprets indexed alpha data.

- `none` — The transparency values of `AlphaData` are between `0` and `1` or are clamped to this range (the default).
- `scaled` — Transform the `AlphaData` to span the portion of the alphamap indicated by the axes `ALim` property, linearly mapping data values to alpha values.
- `direct` — Use the `AlphaData` as indices directly into the alphamap. When not scaled, the data are usually integer values ranging from `1` to `length(alphamap)`. MATLAB maps values less than `1` to the first alpha value in the alphamap, and values greater than `length(alphamap)` to the last alpha value in the alphamap. Values with a decimal portion are fixed to the nearest lower integer. If `AlphaData` is an array of `uint8` integers, then the indexing begins at `0` (that is, MATLAB maps a value of `0` to the first alpha value in the alphamap).

`AmbientStrength`
scalar >= 0 and <= 1

Strength of ambient light. Sets the strength of the ambient light, which is a nondirectional light source that illuminates the entire scene. You must have at least one visible light object in the axes for the ambient light to be visible. The axes `AmbientLightColor` property sets the color of the ambient light, which is therefore the same on all objects in the axes.

You can also set the strength of the diffuse and specular contribution of `light` objects. See the surface `DiffuseStrength` and `SpecularStrength` properties.

`Annotation`

`hg.Annotation` object (read-only)

Handle of `Annotation` object. The `Annotation` property enables you to specify whether this surface object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the surface object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Represent this surface object in a legend (default)
`off`	Do not include this surface object in a legend
`children`	Same as `on` because surface objects do not have children

Setting the IconDisplayStyle property

Set the `IconDisplayStyle` of a graphics object with handle `hobj` to `off`:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Using the IconDisplayStyle property

See "Controlling Legends" for more information and examples.

`BackFaceLighting`
`unlit | lit | {reverselit}`

Face lighting control. Determines how faces are lit when their vertex normals point away from the camera.

- `unlit` — Face not lit.
- `lit` — Face lit in normal way.
- `reverselit` — Face lit as if the vertex pointed towards the camera.

Use this property to discriminate between the internal and external surfaces of an object. See "Back Face Lighting" for an example.

`BeingDeleted`
`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Button press callback function. Executes whenever you press a mouse button while the pointer is over the surface object.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

Set this property to a function handle that references the callback. The function must define at least two input arguments (handle of object associated with the button down event and an event structure, which is empty for this property).
The following example shows how to access the callback object's handle as well as the handle of the figure that contains the object from the callback function.

```
function button_down(src,evnt)
% src - the object that is the source of the event
```

```
% evnt - empty for this property
   sel_typ = get(gcbf,'SelectionType')
   switch sel_typ
      case 'normal'
         disp('User clicked left-mouse button')
         set(src,'Selected','on')
      case 'extend'
         disp('User did a shift-click')
         set(src,'Selected','on')
      case 'alt'
         disp('User did a control-click')
         set(src,'Selected','on')
         set(src,'SelectionHighlight','off')
   end
end
```

Suppose `h` is the handle of a surface object and the `button_down` function is on your MATLAB path. The following statement assigns the `button_down` function to the `ButtonDownFcn` property:

```
set(h,'ButtonDownFcn',@button_down)
```

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`CData`

matrix (of type `double`)

Vertex colors. Values that specify the color at every point in `ZData`.

Mapping CData to a Colormap

You can specify color as indexed values or true color. Indexed color data specifies a single value for each vertex. These values are either scaled to map linearly into the current colormap (see `caxis`) or interpreted directly as indices into the colormap, depending on the setting of the `CDataMapping` property.

CData as True Color

True color defines an RGB value for each vertex. If the coordinate data (`XData`, for example) are contained in `m`-by-`n` matrices, then `CData` must be an `m`-by-`n`-by-`3` array. The first page contains the red components, the second the green components, and the third the blue components of the colors.

Texturemapping the Surface FaceColor

If you set the `FaceColor` property to `texturemap`, `CData` does not need to be the same size as `ZData`, but must be of type `double` or `uint8`. In this case, MATLAB maps `CData` to conform to the surface defined by `ZData`.

`CDataMapping`
`{scaled} | direct`

Direct or scaled color mapping. Determines how MATLAB interprets indexed color data used to color the surface. (If you use true color specification for `CData`, this property has no effect.)

- `scaled` — Transform the color data to span the portion of the colormap indicated by the axes `CLim` property, linearly mapping data values to colors. See the `caxis` reference page for more information on this mapping.
- `direct` — Use the color data as indices directly into the colormap. The color data should then be integer values ranging from 1 to `length(colormap)`. MATLAB maps values less than 1 to the first color in the colormap, and values greater than `length(colormap)` to the last color in the colormap. Values with a decimal portion are fixed to the nearest lower integer.

`Children`
matrix of handles

Always the empty matrix; surface objects have no children.

Surface Properties

`Clipping`
{on} | off

Clipping to axes rectangle. When `Clipping` is `on`, MATLAB does not display any portion of the surface that is outside the axes rectangle.

`CreateFcn`
function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback function executed during object creation. Executes when MATLAB creates a surface object. You must define this property as a default value for surfaces or set the `CreateFcn` property during object creation.

For example, the following statement creates a surface (assuming `x`, `y`, `z`, and `c` are defined), and executes the function referenced by the function handle `@myCreateFcn`.

```
surface(x,y,z,c,'CreateFcn',@myCreateFcn)
```

MATLAB executes this routine after setting all surface properties. Setting this property on an existing surface object has no effect.

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DeleteFcn`
function handle | cell array containing function handle and additional arguments | string (not recommended)

Delete surface callback function. Executes when you delete the surface object (for example, when you issue a `delete` command or clear the axes `cla` or figure `clf`).

For example, the following function displays object property data before the object is deleted.

```
function delete_fcn(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   obj_tp = get(src,'Type');
   disp([obj_tp, ' object deleted'])
   disp('Its user data is:')
   disp(get(src,'UserData'))
end
```

MATLAB executes the function before deleting the object's properties so these values are available to the callback function. The function must define at least two input arguments (handle of object being deleted and an event structure, which is empty for this property).

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DiffuseStrength`

scalar >= 0 and <= 1

Intensity of diffuse light. Sets the intensity of the diffuse component of the light falling on the surface. Diffuse light comes from light objects in the axes. Default value is `0.6`.

You can also set the intensity of the ambient and specular components of the light on the object. See the `AmbientStrength` and `SpecularStrength` properties.

`DisplayName`
string

String used by legend. The `legend` function uses the `DisplayName` property to label the surface object in the legend. The default is an empty string.

- If you specify string arguments with the `legend` function, MATLAB set `DisplayName` to the corresponding string and uses that string for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, MATLAB set `DisplayName` to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add a legend programmatically that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more information and examples.

`EdgeAlpha`
{scalar = `1`} | `flat` | `interp`

Transparency of the surface edges.

- scalar — A single non-NaN scalar value between `0` and `1` that controls the transparency of all the edges of the object. `1` (the default) means fully opaque and `0` means completely transparent.

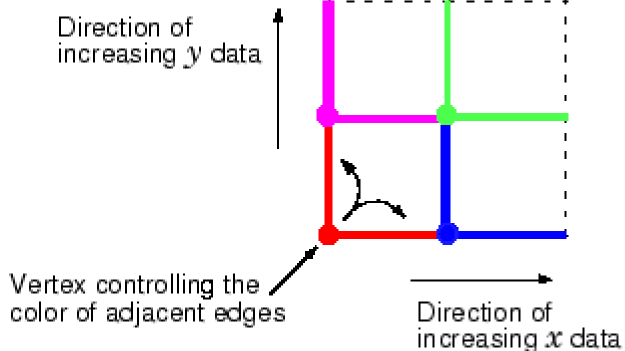
- `flat` — The alpha data (`AlphaData`) value for the first vertex of the face determines the transparency of the edges.
- `interp` — Linear interpolation of the alpha data (`AlphaData`) values at each vertex determines the transparency of the edge.

Note that you must specify `AlphaData` as a matrix equal in size to `ZData` to use `flat` or `interp` `EdgeAlpha`.

`EdgeColor`
`{ColorSpec} | none | flat | interp`

Color of the surface edge. Determines how MATLAB colors the edges of the individual faces that make up the surface.

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for edges. The default value is `[0 0 0]` (`black`). See `ColorSpec` for more information on specifying color.
- `none` — Edges not drawn.
- `flat` — The `CData` value of the first vertex for a face determines the color of each edge.



- `interp` — Linear interpolation of the `CData` values at the face vertices determines the edge color.

`EdgeLighting`
`{none} | flat | gouraud | phong`

Algorithm used for lighting calculations. Selects the algorithm used to calculate the effect of light objects on surface edges.

- `none` — Lights do not affect the edges of this object.
- `flat` — The effect of light objects is uniform across each edge of the surface.
- `gouraud` — The effect of light objects is calculated at the vertices and then linearly interpolated across the edge lines.
- `phong` — The effect of light objects is determined by interpolating the vertex normals across each edge line and calculating the reflectance at each pixel. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

`EraseMode`
`{normal} | none | xor | background`

Erase mode. Controls the technique MATLAB uses to draw and erase surface objects. Alternative erase modes are useful for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the surface when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it because MATLAB stores no information about its former location.

- `xor` — Draw and erase the surface by performing an exclusive OR (XOR) with each pixel index of the screen behind it. Erasing the surface does not damage the color of the objects behind it. However, surface color depends on the color of the screen behind it and is correctly colored only when over the axes background `Color`, or the figure background `Color` if the axes `Color` is `none`.
- `background` — Erase the surface by drawing it in the axes background `Color`, or the figure background `Color` if the axes `Color` is set to `none`. This damages objects that are behind the erased object, but surface objects are always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors (for example, performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`FaceAlpha`
{scalar = `1`} | `flat` | `interp` | `texturemap`

Transparency of the surface faces.

- scalar — A single non-NaN scalar value between `0` and `1` that controls the transparency of all the faces of the object. `1` (the default) means fully opaque and `0` means completely transparent (invisible).

- `flat` — The values of the alpha data (`AlphaData`) determine the transparency for each face. The alpha data at the first vertex determine the transparency of the entire face.
- `interp` — Bilinear interpolation of the alpha data (`AlphaData`) at each vertex determines the transparency of each face.
- `texturemap` — Use transparency for the texture map.

Note that you must specify `AlphaData` as a matrix equal in size to `ZData` to use `flat` or `interp` `FaceAlpha`.

`FaceColor`
`ColorSpec | none | {flat} | interp | texturemap`

Color of the surface face.

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for faces. See the `ColorSpec` reference page for more information on specifying color.
- `none` — Do not draw faces. Note that edges are drawn independently of faces.
- `flat` — The values of `CData` determine the color for each face of the surface. The color data at the first vertex determine the color of the entire face.
- `interp` — Bilinear interpolation of the values at each vertex (the `CData`) determines the coloring of each face.
- `texturemap` — Texture map the `CData` to the surface. MATLAB transforms the color data so that it conforms to the surface. (See the texture mapping example.)

`FaceLighting`
`{none} | flat | gouraud | phong`

Algorithm used for lighting calculations. Selects the algorithm used to calculate the effect of `light` objects on the surface.

- `none` — Lights do not affect the faces of this object.
- `flat` — The effect of light objects is uniform across the faces of the surface. Select this choice to view faceted objects.
- `gouraud` — The effect of light objects is calculated at the vertices and then linearly interpolated across the faces. Select this choice to view curved surfaces.
- `phong` — The effect of light objects is determined by interpolating the vertex normals across each face and calculating the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. This property is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

- `on` — Handles are always visible.
- `callback` — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching

the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

`HitTest`
`{on} | off`

Selectable by mouse click. Determines if the surface can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the surface. If `HitTest` is `off`, clicking on the surface selects the object below it (which might be the axes containing it).

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the `Interruptible` property affects only the callbacks for the`ButtonDownFcn` property. A *running* callback is

the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both the callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted.

For more information, see "Control Callback Execution and Interruption".

`LineStyle`

`{-} | -- | : | -. | none`

Line style of surface edges.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line
'none'	No line

`LineWidth`

size in points

Edge line width. The width of the lines in points used to draw surface edges. The default width is `0.5` points. `1` point = $^1/_{72}$ inch.

`Marker`

character (see table)

Marker symbol. Specifies symbols that are displayed at vertices. You can set values for the `Marker` property independently from the `LineStyle` property. For a list of supported marker symbols, see the following table.

Marker Specifiers Table

Specifier	Marker Type
’+’	Plus sign
’o’	Circle
’*’	Asterisk
’.’	Point
’x’	Cross
’square’ or ’s’	Square
’diamond’ or ’d’	Diamond
’^’	Upward-pointing triangle
’v’	Downward-pointing triangle
’>’	Right-pointing triangle
’<’	Left-pointing triangle
’pentagram’ or ’p’	Five-pointed star (pentagram)
’hexagram’ or ’h''’	Six-pointed star (hexagram)
’none’	No marker (default)

`MarkerEdgeColor`

`none | {auto} | flat | ColorSpec`

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `none` — Specifies no color, which makes nonfilled markers invisible.
- `auto` — Uses same color as the `EdgeColor` property.
- `flat` — Uses the `CData` value of the vertex to determine the color of the marker edge.

- `ColorSpec` — Defines color to use.

`MarkerFaceColor`
`{none} | auto | flat | ColorSpec`

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `none` — Makes the interior of the marker transparent, allowing the background to show through.
- `auto` — Uses the axes `Color` for the marker face color.
- `flat` — Uses the `CData` value of the vertex to determine the color of the face.
- `ColorSpec` — Defines a single color to use for all markers on the surface.

`MarkerSize`
scalar

Marker size. Size of the marker in points. The default value is `6`.

Note MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`MeshStyle`
`{both} | row | column`

Row and column lines. Specifies whether to draw all edge lines or just row or column edge lines.

- `both` — Draws edges for both rows and columns.
- `row` — Draws row edges only.
- `column` — Draws column edges only.

`NormalMode`
`{auto} | manual`

MATLAB generated or user-specified normal vectors.

- `auto` — MATLAB calculates vertex normals based on the coordinate data
- `manual` — If you specify your own vertex normals, MATLAB sets this property to `manual` and does not generate its own data.

See also the `VertexNormals` property.

`Parent`
handle of axes, hggroup, or hgtransform

Parent of surface object. Contains the handle of the surface object's parent. The parent of a surface object is the axes, hggroup, or hgtransform object that contains it.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Is object selected? When this property is `on`, MATLAB displays a dashed bounding box around the surface if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing a dashed bounding box around the surface. When `SelectionHighlight` is `off`, MATLAB does not draw the box.

`SpecularColorReflectance`
scalar in the range `0` to `1`

Color of specularly-reflected light. When this property is `0`, the color of the specularly-reflected light depends on both the color of the object from which it reflects and the color of the light source. When set to `1`, the color of the specularly-reflected light depends only on the color or the light source (that is, the light object `Color` property). The proportions vary linearly for values in between.

`SpecularExponent`
scalar >= 1

Harshness of specular reflection. Controls the size of the specular spot. Most materials have exponents in the range of `5` to `20`.

`SpecularStrength`
scalar >= 0 and <= 1

Intensity of specular light. Sets the intensity of the specular component of the light falling on the surface. Specular light comes from light objects in the axes.

You can also set the intensity of the ambient and diffuse components of the light on the surface object. See the `AmbientStrength` and `DiffuseStrength` properties. Also see the `material` function.

`Tag`
string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

`Type`
string (read-only)

Class of the graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For surface objects, `Type` is always '`surface`'.

`UIContextMenu`
handle of uicontextmenu object

Associate a context menu with the surface. Assign this property the handle of a uicontextmenu object created in the same figure as the surface. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the surface.

`UserData`
matrix

User-specified data. Data you want to associate with the surface object. The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`VertexNormals`
vector | matrix

Surface normal vectors. The vertex normals for the surface. MATLAB generates this data to perform lighting calculations. You can supply your own vertex normal data, even if it does not match the coordinate data. This can be useful to produce interesting lighting effects.

`Visible`
`{on} | off`

Surface object visibility. By default, all surfaces are visible. When set to `off`, the surface is not visible, but still exists, and you can query and set its properties.

`XData`
vector | matrix

X-coordinates. The *x*-position of the surface points. If you specify a row vector, `surface` replicates the row internally until it has the same number of columns as `ZData`.

`YData`
vector | matrix

Y-coordinates. The *y*-position of the surface points. If you specify a row vector, `surface` replicates the row internally until it has the same number of rows as `ZData`.

`ZData`
matrix

Z-coordinates. The *z*-position of the surface data points. See the Description section for more information.

See Also

`surface`

Purpose

Define surfaceplot properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Editor is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

Note that you cannot define default properties for surfaceplot objects.

See Plot Objects for information on surfaceplot objects.

Surfaceplot Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`AlphaData`
: `m-by-n` matrix of `double` or `uint8`

  Transparency data. A matrix of non-NaN values specifying the transparency of each face or vertex of the object. The `AlphaData` can be of class `double` or `uint8`.

  MATLAB software determines the transparency in one of the following ways:

  - Using the elements of `AlphaData` as transparency values (`AlphaDataMapping` set to `none`)
  - Using the elements of `AlphaData` as indices into the current alphamap (`AlphaDataMapping` set to `direct`)
  - Scaling the elements of `AlphaData` to range between the minimum and maximum values of the axes `ALim` property (`AlphaDataMapping` set to `scaled`, the default)

`AlphaDataMapping`
: `none | direct | {scaled}`

Transparency mapping method. Determines how MATLAB interprets indexed alpha data. Values for this property are:

- `none` — The transparency values of `AlphaData` are between `0` and `1` or are clamped to this range.
- `scaled` — Transform the `AlphaData` to span the portion of the alphamap indicated by the axes `ALim` property, linearly mapping data values to alpha values (the default).
- `direct` — Use the `AlphaData` as indices directly into the alphamap. When not scaled, the data are usually integer values ranging from `1` to `length(alphamap)`. MATLAB maps values less than `1` to the first alpha value in the alphamap, and values greater than `length(alphamap)` to the last alpha value in the alphamap. Values with a decimal portion are fixed to the nearest, lower integer. If `AlphaData` is an array of `uint8` integers, then the indexing begins at `0` (that is, MATLAB maps a value of `0` to the first alpha value in the alphamap).

`AmbientStrength`

scalar >= 0 and <= 1

Strength of ambient light. Sets the strength of the ambient light, which is a nondirectional light source that illuminates the entire scene. You must have at least one visible light object in the axes for the ambient light to be visible. The axes `AmbientLightColor` property sets the color of the ambient light, which is therefore the same on all objects in the axes.

You can also set the strength of the diffuse and specular contribution of `light` objects. See the surfaceplot `DiffuseStrength` and `SpecularStrength` properties.

`Annotation`

`hg.Annotation` object (read-only)

Handle of `Annotation` object. The `Annotation` property enables you to specify whether this surfaceplot object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the surfaceplot object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
`on`	Represent this surfaceplot object in a legend (default)
`off`	Do not include this surfaceplot object in a legend
`children`	Same as `on` because surfaceplot objects do not have children

Setting the IconDisplayStyle property

Set the `IconDisplayStyle` of a graphics object with handle `hobj` to `off`:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Using the IconDisplayStyle property

See “Controlling Legends” for more information and examples.

`BackFaceLighting`

`unlit | lit | {reverselit}`

Face lighting control. Determines how faces are lit when their vertex normals point away from the camera.

- `unlit` — Face not lit.
- `lit` — Face lit in normal way.
- `reverselit` — Face lit as if the vertex pointed towards the camera.

Use this property to discriminate between the internal and external surfaces of an object. See "Back Face Lighting" for an example.

`BeingDeleted`

`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`
string | function handle

Button press callback function. Executes whenever you press a mouse button while the pointer is over this object, but not over another graphics object.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

This property can be:

- A string that is a valid MATLAB expression
- The name of a MATLAB file
- A function handle

Set this property to a function handle that references the callback. The expressions execute in the MATLAB workspace.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`CData`

matrix

Vertex colors. A matrix containing values that specify the color at every point in `ZData`. If you set the `FaceColor` property to `texturemap`, `CData` does not need to be the same size as `ZData`. In this case, MATLAB maps `CData` to conform to the surfaceplot defined by `ZData`.

You can specify color as indexed values or true color. Indexed color data specifies a single value for each vertex. These values are either scaled to map linearly into the current colormap (see `caxis`) or interpreted directly as indices into the colormap, depending on the setting of the `CDataMapping` property. Note that any non-texture data passed as an input argument must be of type `double`.

True color defines an RGB value for each vertex. If the coordinate data (`XData`, for example) are contained in `m`-by-`n` matrices, then `CData` must be an `m`-by-`n`-by-`3` array. The first page contains the red components, the second the green components, and the third the blue components of the colors.

`CDataMapping`

`{scaled} | direct`

Direct or scaled color mapping. Determines how MATLAB interprets indexed color data used to color the surfaceplot. (If you use true color specification for `CData`, this property has no effect.)

- `scaled` — Transform the color data to span the portion of the colormap indicated by the axes `CLim` property, linearly mapping data values to colors. See the `caxis` reference page for more information on this mapping.
- `direct` — Use the color data as indices directly into the colormap. The color data should then be integer values ranging from 1 to `length(colormap)`. MATLAB maps values less than

1 to the first color in the colormap, and values greater than `length(colormap)` to the last color in the colormap. Values with a decimal portion are fixed to the nearest lower integer.

`CDataMode`
`{auto} | manual`

Use automatic or user-specified color data values. If you specify `CData`, MATLAB sets this property to `manual` and uses the `CData` values to color the surfaceplot.

If you set `CDataMode` to `auto` after having specified `CData`, MATLAB resets the color data of the surfaceplot to that defined by `ZData`, overwriting any previous values for `CData`.

`CDataSource`
string (MATLAB variable)

Link CData to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `CData`.

MATLAB reevaluates this property only when you set it. Therefore, a change to workspace variables appearing in an expression does not change `CData`.

You can use the `refreshdata` function to force an update of the object's data. `refreshdata` also enables you to specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`. See the `refreshdata` reference page for more information.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`Children`
: matrix of handles

  Always the empty matrix; surfaceplot objects have no children.

`Clipping`
: `{on} | off`

  Clipping to axes rectangle. When `Clipping` is `on`, MATLAB does not display any portion of the surfaceplot that is outside the axes rectangle.

`CreateFcn`
: function handle | cell array containing function handle and additional arguments | string (not recommended)

  Callback function executed during object creation. Defines a callback that executes when MATLAB creates an object. The default is an empty array.

  You must specify the callback during the creation of the object. For example:

  ```
  graphicfcn(y,'CreateFcn',@CallbackFcn)
  ```

  where `@CallbackFcn` is a function handle that references the callback function and `graphicfcn` is the plotting function which creates this object.

MATLAB executes this routine after setting all other object properties. Setting this property on an existing object has no effect.

The handle of the object whose `CreateFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DeleteFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback executed during object deletion. Executes when this object is deleted (for example, this might happen when you issue a `delete` command on the object, its parent axes, or the figure containing it). MATLAB executes the callback before destroying the object's properties so the callback routine can query these values. The default is an empty array.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which can be queried using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

See the `BeingDeleted` property for related information.

`DiffuseStrength`

scalar >= 0 and <= 1

Intensity of diffuse light. Sets the intensity of the diffuse component of the light falling on the surface. Diffuse light comes from light objects in the axes. Default value is `0.6`.

You can also set the intensity of the ambient and specular components of the light on the object. See the `AmbientStrength` and `SpecularStrength` properties.

`DisplayName`
string

String used by legend. The `legend` function uses the `DisplayName` property to label the surfaceplot object in the legend. The default is an empty string.

- If you specify string arguments with the `legend` function, MATLAB set `DisplayName` to the corresponding string and uses that string for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, MATLAB set `DisplayName` to the edited string.
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add a legend programmatically that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more information and examples.

`EdgeAlpha`
{scalar = `1`} | `flat` | `interp`

Transparency of the patch and surface edges.

- scalar — A single non-NaN scalar value between `0` and `1` that controls the transparency of all the edges of the object. `1` (the default) means fully opaque and `0` means completely transparent.

- `flat` — The alpha data (`AlphaData`) value for the first vertex of the face determines the transparency of the edges.
- `interp` — Linear interpolation of the alpha data (`AlphaData`) values at each vertex determines the transparency of the edge.

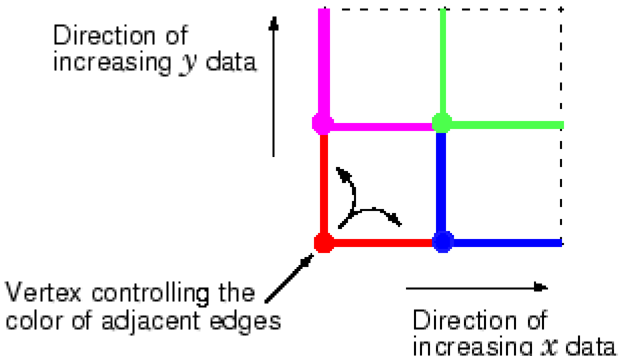
Note that you must specify `AlphaData` as a matrix equal in size to `ZData` to use `flat` or `interp` `EdgeAlpha`.

Note that you must specify `AlphaData` as a matrix equal in size to `ZData` to use `flat` or `interp` `EdgeAlpha`.

`EdgeColor`
`{ColorSpec} | none | flat | interp`

Color of the surfaceplot edge. This property determines how MATLAB colors the edges of the individual faces that make up the surface:

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for edges. The default `EdgeColor` is black. See `ColorSpec` for more information on specifying color.
- `none` — Edges are not drawn.
- `flat` — The `CData` value of the first vertex for a face determines the color of each edge.



- `interp` — Linear interpolation of the `CData` values at the face vertices determines the edge color.

`EdgeLighting`
`{none} | flat | gouraud | phong`

Algorithm used for lighting calculations. Selects the algorithm used to calculate the effect of light objects on surfaceplot edges.

- `none` — Lights do not affect the edges of this object.
- `flat` — The effect of light objects is uniform across each edge of the surfaceplot.
- `gouraud` — The effect of light objects is calculated at the vertices and then linearly interpolated across the edge lines.
- `phong` — The effect of light objects is determined by interpolating the vertex normals across each edge line and calculating the reflectance at each pixel. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

`EraseMode`
`{normal} | none | xor | background`

Erase mode. Controls the technique MATLAB uses to draw and erase objects. Use alternative erase modes for creating animated sequences, where control of the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to correctly render all objects. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the object when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it because MATLAB stores no information about its former location.
- `xor` — Draw and erase the object by performing an exclusive OR (XOR) with the color of the screen beneath it. This mode does not damage the color of the objects beneath it. However, the object's color depends on the color of whatever is beneath it on the display.
- `background` — Erase the object by redrawing it in the axes background `Color`, or the figure background `Color` if the axes `Color` property is `none`. This damages objects that are behind the erased object, but properly colors the erased object.

Set the axes background color with the axes `Color` property. Set the figure background color with the figure `Color` property.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors (for example, performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting

to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`FaceAlpha`

{scalar = `1`} | `flat` | `interp` | `texturemap`

Transparency of the surfaceplot faces. This property can be any of the following:

- scalar — A single non-NaN scalar value between `0` and `1` that controls the transparency of all the faces of the object. `1` (the default) means fully opaque and `0` means completely transparent (invisible).
- `flat` — The values of the alpha data (`AlphaData`) determine the transparency for each face. The alpha data at the first vertex determine the transparency of the entire face.
- `interp` — Bilinear interpolation of the alpha data (`AlphaData`) at each vertex determines the transparency of each face.
- `texturemap` — Use transparency for the texture map.

Note that you must specify `AlphaData` as a matrix equal in size to `ZData` to use `flat` or `interp` `FaceAlpha`.

`FaceColor`

`ColorSpec` | `none` | `{flat}` | `interp`

Color of the surfaceplot face. This property can be any of the following:

- `ColorSpec` — A three-element RGB vector or one of the MATLAB predefined names, specifying a single color for faces. See `ColorSpec` for more information on specifying color.

- `none` — Do not draw faces. Note that edges are drawn independently of faces.
- `flat` — The values of `CData` determine the color for each face of the surface. The color data at the first vertex determine the color of the entire face.
- `interp` — Bilinear interpolation of the values at each vertex (the `CData`) determines the coloring of each face.
- `texturemap` — Texture map the `CData` to the surface. MATLAB transforms the color data so that it conforms to the surface. (See the texture mapping example for `surface`.)

`FaceLighting`
`{none} | flat | gouraud | phong`

Algorithm used for lighting calculations. This property selects the algorithm used to calculate the effect of light objects on the surface. Choices are

- `none` — Lights do not affect the faces of this object.
- `flat` — The effect of light objects is uniform across the faces of the surface. Select this choice to view faceted objects.
- `gouraud` — The effect of light objects is calculated at the vertices and then linearly interpolated across the faces. Select this choice to view curved surfaces.
- `phong` — The effect of light objects is determined by interpolating the vertex normals across each face and calculating the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. `HandleVisibility`

is useful for preventing command-line users from accidentally accessing objects that you need to protect for some reason.

- `on` — Handles are always visible.
- `callback` — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

Functions Affected by Handle Visibility

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

Properties Affected by Handle Visibility

When a handle's visibility is restricted using `callback` or `off`, the object's handle does not appear in its parent's `Children` property, figures do not appear in the root's `CurrentFigure` property, objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property, and axes do not appear in their parent's `CurrentAxes` property.

Overriding Handle Visibility

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties. See also `findall`.

Handle Validity

Hidden handles are still valid. If you know an object's handle, you can `set` and `get` its properties and pass it to any function that operates on handles.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`HitTest`
`{on} | off`

Selectable by mouse click. Determines whether this object can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the surface. If `HitTest` is `off`, clicking this object selects the object below it (which is usually the axes containing it).

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the `Interruptible` property affects only the callbacks for the`ButtonDownFcn` property. A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both the callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see "Control Callback Execution and Interruption".

`LineStyle`

`{-} | -- | :  | -.  | none`

Line style of surfaceplot object.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line
'none'	No line

Use `LineStyle` `none` when you want to place a marker at each point but do not want the points connected with a line (see the `Marker` property).

`LineWidth`
size in points

Width of linear objects and edges of filled areas. Specify in points. `1` point = $^1/_{72}$ inch. The default is `0.5` points.

`Marker`
character (see table)

Marker symbol. Specifies marks that display at data points. You can set values for the `Marker` property independently from the `LineStyle` property. For a list of supported marker symbols, see the following table.

Marker Specifiers Table

Specifier	Marker Type
'+'	Plus sign
'o'	Circle

Specifier	Marker Type
'*'	Asterisk
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h''	Six-pointed star (hexagram)
'none'	No marker (default)

MarkerEdgeColor
none | {auto} | flat | ColorSpec

Marker edge color. The color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).

- none — Specifies no color, which makes nonfilled markers invisible.
- auto — Uses same color as the EdgeColor property.
- flat — Uses the CData value of the vertex to determine the color of the marker edge.
- ColorSpec — Defines color to use.

MarkerFaceColor
{none} | auto | flat | ColorSpec

Marker face color. The fill color for markers that are closed shapes (circle, square, diamond, pentagram, hexagram, and the four triangles).

- `none` — Makes the interior of the marker transparent, allowing the background to show through.
- `auto` — Uses the axes `Color` for the marker face color.
- `flat` — Uses the `CData` value of the vertex to determine the color of the face.
- `ColorSpec` — Defines a single color to use for all markers on the surfaceplot.

`MarkerSize`
scalar

Marker size. Size of the marker in points. The default value is `6`.

Note MATLAB draws the point marker (specified by the `'.'` symbol) at one-third the specified size.

`MeshStyle`
`{both} | row | column`

Row and column lines. Specifies whether to draw all edge lines or just row or column edge lines.

- `both` — Draws edges for both rows and columns.
- `row` — Draws row edges only.
- `column` — Draws column edges only.

`NormalMode`
`{auto} | manual`

MATLAB generated or user-specified normal vectors.

- `auto` — MATLAB calculates vertex normals based on the coordinate data
- `manual` — If you specify your own vertex normals, MATLAB sets this property to `manual` and does not generate its own data.

See also the `VertexNormals` property.

`Parent`
handle of parent axes, hggroup, or hgtransform

Parent of object. Handle of the object's parent. The parent is normally the axes, hggroup, or hgtransform object that contains the object.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Selected`
`on | {off}`

Object selection state. When you set this property to `on`, MATLAB displays selection handles at the corners and midpoints if the `SelectionHighlight` property is also `on` (the default). You can, for example, define the `ButtonDownFcn` callback to set this property to `on`, thereby indicating that this particular object is selected. This property is also set to `on` when an object is manually selected in plot edit mode.

`SelectionHighlight`
`{on} | off`

Selected objects highlighted.

- `on` — MATLAB indicates the selected state by drawing a dashed bounding box around the surface.
- `off` — MATLAB does not draw the bounding box except when in plot edit mode and objects are selected manually.

`SpecularColorReflectance`

scalar in the range `0` to `1`

Color of specularly-reflected light. When this property is `0`, the color of the specularly-reflected light depends on both the color of the object from which it reflects and the color of the light source. When set to `1`, the color of the specularly-reflected light depends only on the color or the light source (that is, the light object `Color` property). The proportions vary linearly for values in between.

`SpecularExponent`

scalar >= 1

Harshness of specular reflection. Controls the size of the specular spot. Most materials have exponents in the range of `5` to `20`.

`SpecularStrength`

scalar >= 0 and <= 1

Intensity of specular light. Sets the intensity of the specular component of the light falling on the surfaceplot. Specular light comes from light objects in the axes.

You can also set the intensity of the ambient and diffuse components of the light on the surfaceplot object. See the `AmbientStrength` and `DiffuseStrength` properties. Also see the `material` function.

`Tag`

string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

For example, create an areaseries object and set the `Tag` property.

```
t = area(Y,'Tag','area1')
```

When you want to access objects of a given type, use `findobj` to find the object's handle. The following statement changes the `FaceColor` property of the object whose `Tag` is `area1`.

```
set(findobj('Tag','area1'),'FaceColor','red')
```

`Type`
string (read only)

Class of the graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For surfaceplot objects, `Type` is always the string `'surface'`.

`UIContextMenu`
handle of uicontextmenu object

Associate context menu with object. Handle of a uicontextmenu object created in the object's parent figure. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the object. The default value is an empty array.

`UserData`
array

User-specified data. Data you want to associate with this object (including cell arrays and structures). The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`VertexNormals`
vector | matrix

Surfaceplot normal vectors. Contains the vertex normals for the surfaceplot. MATLAB generates this data to perform lighting calculations. You can supply your own vertex normal data, even

if it does not match the coordinate data. This can be useful to produce interesting lighting effects.

Visible
`{on} | off`

Visibility of object and its children.

- `on` — Object and all children of the object are visible unless the child object's `Visible` property is `off`.
- `off` — Object not displayed. However, the object still exists and you can set and query its properties.

XData
vector | matrix

X-coordinates. The *x*-position of the surfaceplot data points. If you specify a row vector, MATLAB replicates the row internally until it has the same number of columns as `ZData`.

XDataMode
`{auto} | manual`

Use automatic or user-specified x-axis values. If you specify `XData` (by setting the `XData` property or specifying the `X` input argument), MATLAB sets this property to `manual` and uses the specified values to label the *x*-axis.

If you set `XDataMode` to `auto` after specifying `XData`, MATLAB resets the *x*-axis ticks to `1:size(YData,1)` or to the column indices of the `ZData`, overwriting any previous values for `XData`.

XDataSource
MATLAB variable, as a string

Link `XData` to MATLAB variable. Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `XData`. The default value is an empty array.

```
set(h,'XDataSource','xdatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `XDataSource` does not change the object's `XData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`YData`
vector | matrix

Y-coordinates. The *y*-position of the surfaceplot data points. If you specify a row vector, MATLAB replicates the row internally until it has the same number of rows as `ZData`.

`YDataMode`
{auto} | manual

Use automatic or user-specified y-axis values. If you specify `XData`, MATLAB sets this property to `manual`.

If you set `YDataMode` to `auto` after having specified `YData`, MATLAB resets the *y*-axis ticks and *y*-tick labels to the row indices of the `ZData`, overwriting any previous values for `YData`.

`YDataSource`
MATLAB variable, as a string

Link `YData` *to MATLAB variable.* Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `YData`. The default value is an empty array.

```
set(h,'YDataSource','Ydatavariablename')
```

MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's `YDataSource` does not change the object's `YData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

`ZData`
matrix

Z-coordinates. The *z*-position of the surfaceplot data points. See the Description section for more information.

`ZDataSource`
MATLAB variable, as a string

Link `ZData` *to MATLAB variable.* Set this property to a MATLAB variable that is evaluated in the base workspace to generate the `ZData`. The default value is an empty array.

```
set(h,'ZDataSource','zdatavariablename')
```

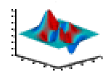
MATLAB requires a call to `refreshdata` when you set this property. Changing workspace variables used as an object's

`ZDataSource` does not change the object's `ZData` values, but you can use `refreshdata` to force an update of the object's data. `refreshdata` also lets you specify that the data source variable be evaluated in the workspace of a function from which you call `refreshdata`.

Note If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Purpose

Surface plot with colormap-based lighting



Syntax

```
surfl(Z)
surfl(...,'light')
surfl(...,s)
surfl(X,Y,Z,s,k)
h = surfl(...)
```

Description

The `surfl` function displays a shaded surface based on a combination of ambient, diffuse, and specular lighting models.

`surfl(Z)` and surfl(X,Y,Z) create three-dimensional shaded surfaces using the default direction for the light source and the default lighting coefficients for the shading model. `X`, `Y`, and `Z` are vectors or matrices that define the *x*, *y*, and *z* components of a surface.

`surfl(...,'light')` produces a colored, lighted surface using a MATLAB light object. This produces results different from the default lighting method, `surfl(...,'cdata')`, which changes the color data for the surface to be the reflectance of the surface.

`surfl(...,s)` specifies the direction of the light source. `s` is a two- or three-element vector that specifies the direction from a surface to a light source. `s = [sx sy sz]` or `s = [azimuth elevation]`. The default `s` is 45° counterclockwise from the current view direction.

`surfl(X,Y,Z,s,k)` specifies the reflectance constant. `k` is a four-element vector defining the relative contributions of ambient light, diffuse reflection, specular reflection, and the specular shine coefficient. `k = [ka kd ks shine]` and defaults to `[.55,.6,.4,10]`.

`h = surfl(...)` returns a handle to a surface graphics object.

Tips

surfl does not accept complex inputs.

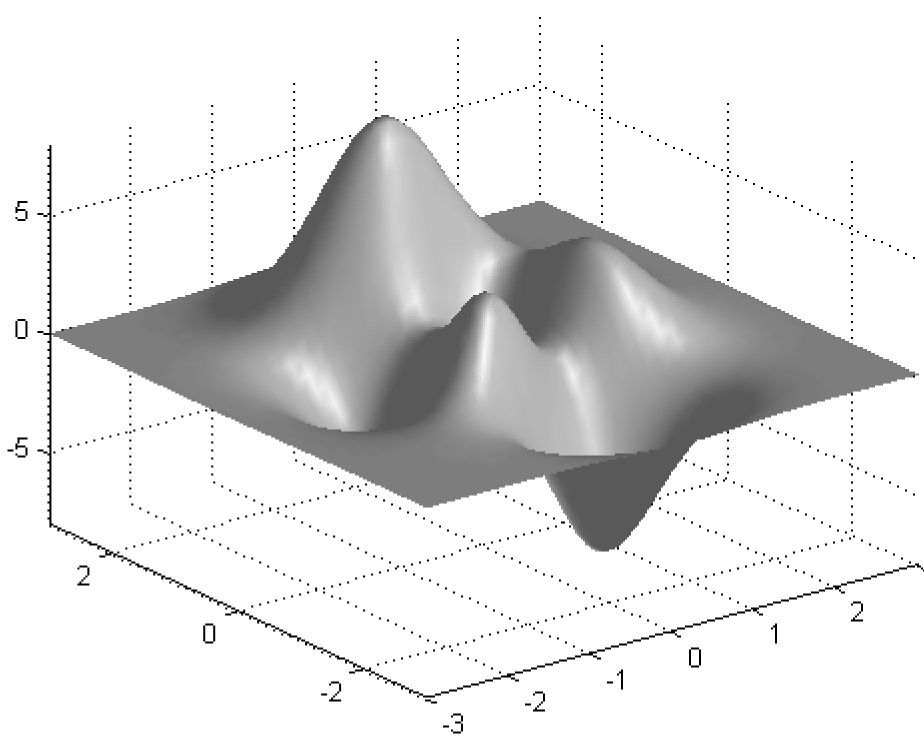
For smoother color transitions, use colormaps that have linear intensity variations (e.g., gray, copper, bone, pink).

The ordering of points in the X, Y, and Z matrices defines the inside and outside of parametric surfaces. If you want the opposite side of the surface to reflect the light source, use surfl(X',Y',Z'). Because of the way surface normal vectors are computed, surfl requires matrices that are at least 3-by-3.

Examples

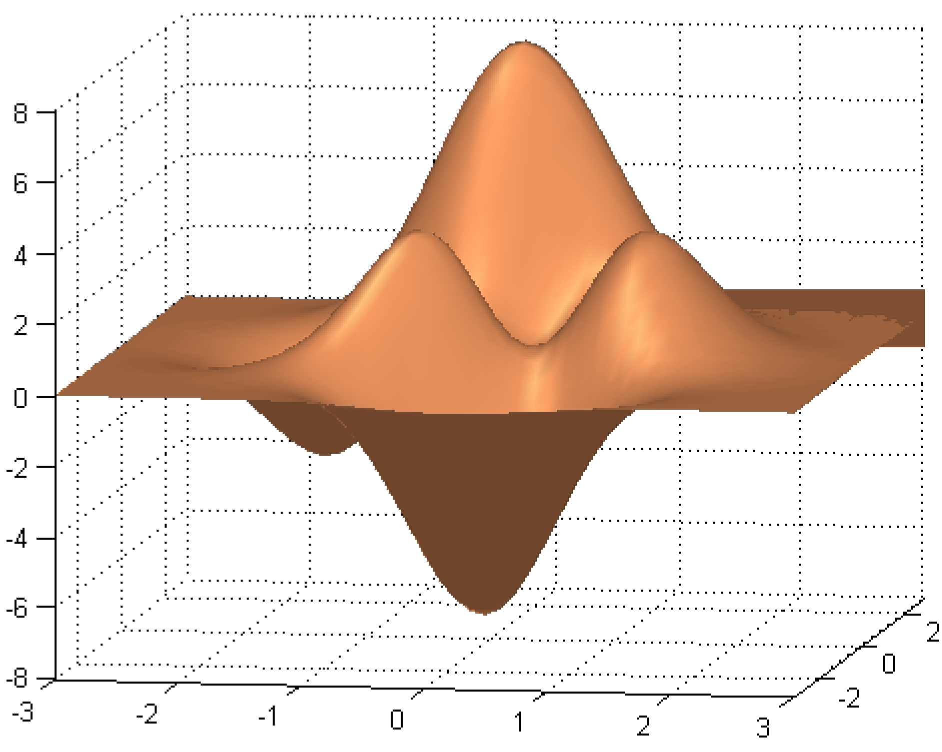
View peaks using colormap-based lighting.

```
[x,y] = meshgrid(-3:1/8:3);
z = peaks(x,y);
surfl(x,y,z);
shading interp
colormap(gray);
axis([-3,3,-3,3,-8,8])
```



To plot a lighted surface from a view direction other than the default,

```
view([10 10])
grid on
hold on
surfl(peaks)
shading interp
colormap copper
hold off
```



See Also colormap | shading | light

Purpose

Compute and display 3-D surface normals

Syntax

```
surfnorm(Z)
surfnorm(X,Y,Z)
surfnorm(axes_handle,___)
surfnorm(___,Name,Value)
[Nx,Ny,Nz] = surfnorm(___)
```

Description

`surfnorm(Z)` plots a surface of the matrix `Z` with `surf` and displays its surface normals as radiating vectors.

`surfnorm(X,Y,Z)` plots a surface and its surface normals from the vectors or matrices `X`, `Y`, and matrix `Z`. `X`, `Y`, and `Z` must be the same size.

`surfnorm(axes_handle,___)` plots into `axes_handle` instead of `gca` and it can include any of the input arguments in previous syntaxes.

`surfnorm(___,Name,Value)` can be used to set the value of the specified `Surface Properties`.

`[Nx,Ny,Nz] = surfnorm(___)` returns the components of the 3-D surface normals for the surface without plotting the surface or surface normals.

Tips

- `surfnorm` does not accept complex inputs.
- Reverse the direction of the normals by calling `surfnorm` with transposed arguments:

  ```
  surfnorm(X',Y',Z')
  ```

- The surface normals represent conditions at vertices and are not normalized. Normals for surface elements that face away from the viewer do not display.
- `surfl` uses `surfnorm` to compute surface normals when calculating the reflectance of a surface.

Input Arguments

Z

2–D array of real numbers representing a surface

X

2–D array of real numbers that defines the *x* component of the surface grid

Y

2–D array of real numbers that defines the *y* component of the surface grid

axes_handle

Handle to the target axes in which to plot the surface

If you do not specify `axes_handle`, MATLAB uses current axes.

Name,Value

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

Property names and values of the surface object

See `Surface Properties` for description of property names and values.

Output Arguments

[Nx,Ny,Nz]

x, *y*, and *z* components of the three-dimensional surface normals for the surface

Definitions

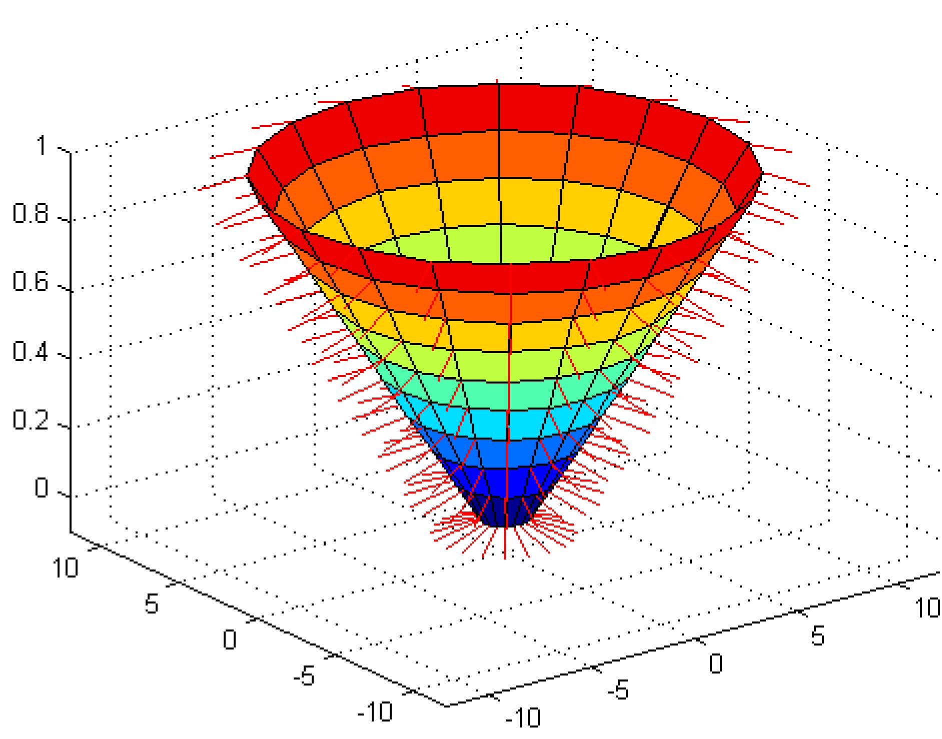
Surface Normal

An imaginary line perpendicular to a flat surface or perpendicular to the tangent plane at a point on a non-flat surface

Examples

Plot the normal vectors for a truncated cone.

```
[x,y,z] = cylinder(1:10);
surfnorm(x,y,z)
axis([-12 12 -12 12 -0.1 1])
```

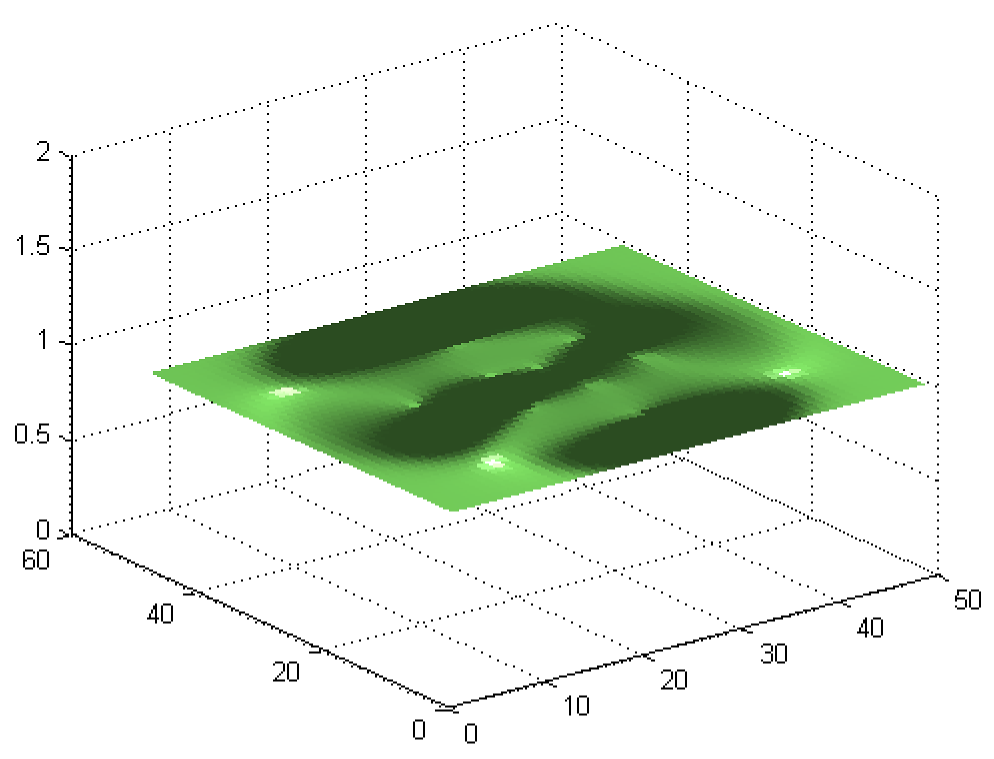


Get normal vectors of an expression representing a surface.

```
[nx, ny, nz] = surfnorm(peaks);
```

You can use these normals in conjunction with `VertexNormals` property and to apply lighting variations to a plain surface as,

```
b = reshape([nx ny nz], 49,49,3);
surf(ones(49), 'VertexNormals', b);%Create a plain surface of size equal
shading interp % remove edge definition from the image
camlight
```



Algorithms

After performing a bicubic fit of the data in the x, y, and z directions, diagonal vectors are computed and crossed to form the normal at each vertex.

See Also

`surf` | `quiver3` | `surface` | `isonormals` | `surfl`

Purpose

Singular value decomposition

Syntax

```
s = svd(X)
[U,S,V] = svd(X)
[U,S,V] = svd(X,0)
[U,S,V] = svd(X,'econ')
```

Description

The `svd` command computes the matrix singular value decomposition.

`s = svd(X)` returns a vector of singular values.

`[U,S,V] = svd(X)` produces a diagonal matrix `S` of the same dimension as `X`, with nonnegative diagonal elements in decreasing order, and unitary matrices `U` and `V` so that `X = U*S*V'`.

`[U,S,V] = svd(X,0)` produces the "economy size" decomposition. If `X` is m-by-n with m > n, then `svd` computes only the first `n` columns of `U` and `S` is n-by-n.

`[U,S,V] = svd(X,'econ')` also produces the "economy size" decomposition. If `X` is m-by-n with m >= n, it is equivalent to `svd(X,0)`. For m < n, only the first m columns of `V` are computed and `S` is m-by-m.

Examples

For the matrix

```
X =
     1     2
     3     4
     5     6
     7     8
```

the statement

```
[U,S,V] = svd(X)
```

produces

```
U =
   -0.1525   -0.8226   -0.3945   -0.3800
```

```
    -0.3499   -0.4214    0.2428    0.8007
    -0.5474   -0.0201    0.6979   -0.4614
    -0.7448    0.3812   -0.5462    0.0407

S =
     14.2691         0
           0    0.6268
           0         0
           0         0

V =
    -0.6414     0.7672
    -0.7672    -0.6414
```

The economy size decomposition generated by

```
[U,S,V] = svd(X,0)
```

produces

```
U =
    -0.1525   -0.8226
    -0.3499   -0.4214
    -0.5474   -0.0201
    -0.7448    0.3812
S =
    14.2691         0
          0    0.6268
V =
    -0.6414    0.7672
    -0.7672   -0.6414
```

Diagnostics

If the limit of 75 QR step iterations is exhausted while seeking a singular value, this message appears:

```
Solution will not converge.
```

svds

Purpose

Find singular values and vectors

Syntax

```
s = svds(A)
s = svds(A,k)
s = svds(A,k,sigma)
s = svds(A,k,'L')
s = svds(A,k,sigma,options)
[U,S,V] = svds(A,...)
[U,S,V,flag] = svds(A,...)
```

Description

`s = svds(A)` computes the six largest singular values and associated singular vectors of matrix `A`. If `A` is `m`-by-`n`, `svds(A)` manipulates eigenvalues and vectors returned by `eigs(B)`, where `B = [sparse(m,m) A; A' sparse(n,n)]`, to find a few singular values and vectors of `A`. The positive eigenvalues of the symmetric matrix `B` are the same as the singular values of `A`.

`s = svds(A,k)` computes the `k` largest singular values and associated singular vectors of matrix `A`.

`s = svds(A,k,sigma)` computes the `k` singular values closest to the scalar shift sigma. For example, `s = svds(A,k,0)` computes the `k` smallest singular values and associated singular vectors.

`s = svds(A,k,'L')` computes the `k` largest singular values (the default).

`s = svds(A,k,sigma,options)` sets some parameters (see `eigs`):

Option Structure Fields and Descriptions

Field name	Parameter	Default
`options.tol`	Convergence tolerance: `norm(AV-US,1)<=tol*norm(A,1)`	`1e-10`
`options.maxit`	Maximum number of iterations	`300`
`options.disp`	Number of values displayed each iteration	`0`

svds checks the accuracy of the computed singular vectors. If the vectors they are not accurate enough returns fewer singular values than requested. To obtain the requested number of singular values, try decreasing the error tolerance in the options structure.

[U,S,V] = svds(A,...) returns three output arguments, and if A is m-by-n:

- U is m-by-k with orthonormal columns
- S is k-by-k diagonal
- V is n-by-k with orthonormal columns
- U*S*V' is the closest rank k approximation to A

[U,S,V,flag] = svds(A,...) returns a convergence flag. If eigs converged, then norm(A*V-U*S,1) <= tol*norm(A,1) and flag is 0. If eigs did not converge, then flag is 1.

Note svds is best used to find a few singular values of a large, sparse matrix. To find all the singular values of such a matrix, svd(full(A)) will usually perform better than svds(A,min(size(A))).

Algorithms

svds(A,k) uses eigs to find the k largest magnitude eigenvalues and corresponding eigenvectors of B = [0 A; A' 0].

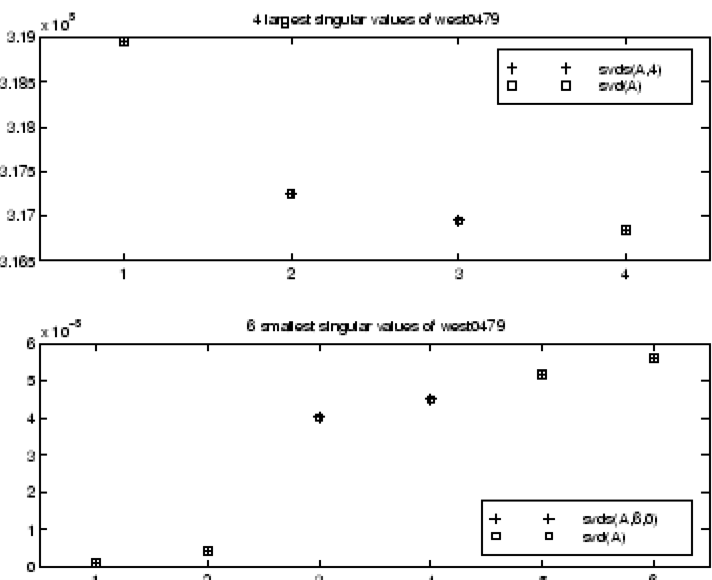
svds(A,k,0) uses eigs to find the 2k smallest magnitude eigenvalues and corresponding eigenvectors of B = [0 A; A' 0], and then selects the k positive eigenvalues and their eigenvectors.

Examples

west0479 is a real 479-by-479 sparse matrix. svd calculates all 479 singular values. svds picks out the largest and smallest singular values.

```
load west0479
s = svd(full(west0479))
sl = svds(west0479,4)
ss = svds(west0479,6,0)
```

These plots show some of the singular values of `west0479` as computed by `svd` and `svds`.



The largest singular value of `west0479` can be computed a few different ways:

```
svds(west0479,1) =
 3.189517598808622e+05
max(svd(full(west0479))) =
 3.18951759880862e+05
norm(full(west0479)) =
 3.189517598808623e+05
```

and estimated:

```
normest(west0479) =
 3.189385666549991e+05
```

See Also

`svd` | `eigs`

Purpose

Swap byte ordering

Syntax

```
Y = swapbytes(X)
```

Description

`Y = swapbytes(X)` reverses the byte ordering of each element in array `X`, converting little-endian values to big-endian (and vice versa). The input array must contain all full, noncomplex, numeric elements.

Examples

Example 1

Reverse the byte order for a scalar 32-bit value, changing hexadecimal `12345678` to `78563412`:

```
A = uint32(hex2dec('12345678'));

B = dec2hex(swapbytes(A))
B =
   78563412
```

Example 2

Reverse the byte order for each element of a 1-by-4 matrix:

```
X = uint16([0 1 128 65535])
X =
      0      1    128  65535

Y = swapbytes(X);
Y =
      0    256  32768  65535
```

Examining the output in hexadecimal notation shows the byte swapping:

```
format hex

X, Y
X =
   0000   0001   0080   ffff
```

```
Y =
   0000   0100   8000   ffff
```

Example 3

Create a three-dimensional array A of 16-bit integers and then swap the bytes of each element:

```
format hex

A = uint16(magic(3) * 150);
A(:,:,2) = A * 40;

A
A(:,:,1) =
   04b0   0096   0384
   01c2   02ee   041a
   0258   0546   012c
A(:,:,2) =
   bb80   1770   8ca0
   4650   7530   a410
   5dc0   d2f0   2ee0

swapbytes(A)
ans(:,:,1) =
   b004   9600   8403
   c201   ee02   1a04
   5802   4605   2c01
ans(:,:,2) =
   80bb   7017   a08c
   5046   3075   10a4
   c05d   f0d2   e02e
```

See Also

typecast

switch/case/otherwise

Purpose

Switch among several cases based on expression

Syntax

```
switch switch_expression
   case case_expression
      statements
   case case_expression
      statements
    :
   otherwise
      statements
end
```

Description

A `switch` block conditionally executes one set of statements from several choices. Each choice is a `case`.

An evaluated *switch_expression* is a scalar or string. An evaluated *case_expression* is a scalar, a string, or a cell array of scalars or strings. The `switch` block tests each case until one of the cases is `true`. A case is `true` when:

- For numbers, `eq(`*case_expression*`,`*switch_expression*`)`.
- For strings, `strcmp(`*case_expression*`,`*switch_expression*`)`.
- For objects that support the `eq` function, `eq(`*case_expression*`,`*switch_expression*`)`.
- For a cell array *case_expression*, at least one of the elements of the cell array matches *switch_expression*, as defined above for numbers, strings, and objects.

When a case is true, MATLAB executes the corresponding statements, and then exits the `switch` block.

`otherwise` is optional, and executes only when no case is true.

Tips

- Unlike the C language `switch` statement, the MATLAB `switch` does not fall through. If the first `case` statement is `true`, the other `case`

statements do not execute. Do not use a `break` statement within a `switch` block.

- A *`case_expression`* cannot include relational operators such as `<` or `>` to compare against the *`switch_expression`*. To test for inequality, use `if-elseif` statements.
- Because MATLAB executes only one case of any `switch` statement, variables defined within one `case` are not available for other cases. For example, if your current workspace does not contain a variable `x`, only cases that define `x` can use it:

```
switch choice
   case 1
      x = -pi:0.01:pi;
   case 2
      % does not know anything about x
end
```

Examples

Conditionally display different text depending on a value entered at the command line:

```
mynumber = input('Enter a number:');

switch mynumber
    case -1
        disp('negative one');
    case 0
        disp('zero');
    case 1
        disp('positive one');
    otherwise
        disp('other value');
end
```

Decide which plot to create based on the value of the string `plottype`:

```
x = [12, 64, 24];
plottype = 'pie3';

switch plottype
    case 'bar'
        bar(x)
        title('Bar Graph')
    case {'pie','pie3'}
        pie3(x)
        title('Pie Chart')
        legend('First','Second','Third')
    otherwise
        warning('Unexpected plot type. No plot created.');
end
```

In MATLAB `switch` blocks, only the first matching case executes:

```
result = 52;

switch(result)
   case 52
      disp('result is 52')
   case {52, 78}
      disp('result is 52 or 78')
end
```

This code returns

```
result is 52
```

See Also

`end` | `for` | `if` | `while`

symamd

Purpose

Symmetric approximate minimum degree permutation

Syntax

```
p = symamd(S)
p = symamd(S,knobs)
[p,stats] = symamd(...)
```

Description

`p = symamd(S)` for a symmetric positive definite matrix `S`, returns the permutation vector `p` such that `S(p,p)` tends to have a sparser Cholesky factor than `S`. To find the ordering for `S`, `symamd` constructs a matrix `M` such that `spones(M'*M) = spones (S)`, and then computes `p = colamd(M)`. The `symamd` function may also work well for symmetric indefinite matrices.

`S` must be square; only the strictly lower triangular part is referenced.

`p = symamd(S,knobs)` where `knobs` is a scalar. If `S` is `n`-by-`n`, rows and columns with more than `knobs*n` entries are removed prior to ordering, and ordered last in the output permutation `p`. If the `knobs` parameter is not present, then `knobs = spparms('wh_frac')`.

`[p,stats] = symamd(...)` produces the optional vector `stats` that provides data about the ordering and the validity of the matrix `S`.

`stats(1)`	Number of dense or empty rows ignored by `symamd`
`stats(2)`	Number of dense or empty columns ignored by `symamd`
`stats(3)`	Number of garbage collections performed on the internal data structure used by `symamd` (roughly of size `8.4*nnz(tril(S,-1)) + 9n` integers)
`stats(4)`	`0` if the matrix is valid, or `1` if invalid
`stats(5)`	Rightmost column index that is unsorted or contains duplicate entries, or `0` if no such column exists
`stats(6)`	Last seen duplicate or out-of-order row index in the column index given by `stats(5)`, or `0` if no such row index exists
`stats(7)`	Number of duplicate and out-of-order row indices

Although, MATLAB built-in functions generate valid sparse matrices, a user may construct an invalid sparse matrix using the MATLAB C or Fortran APIs and pass it to `symamd`. For this reason, `symamd` verifies that `S` is valid:

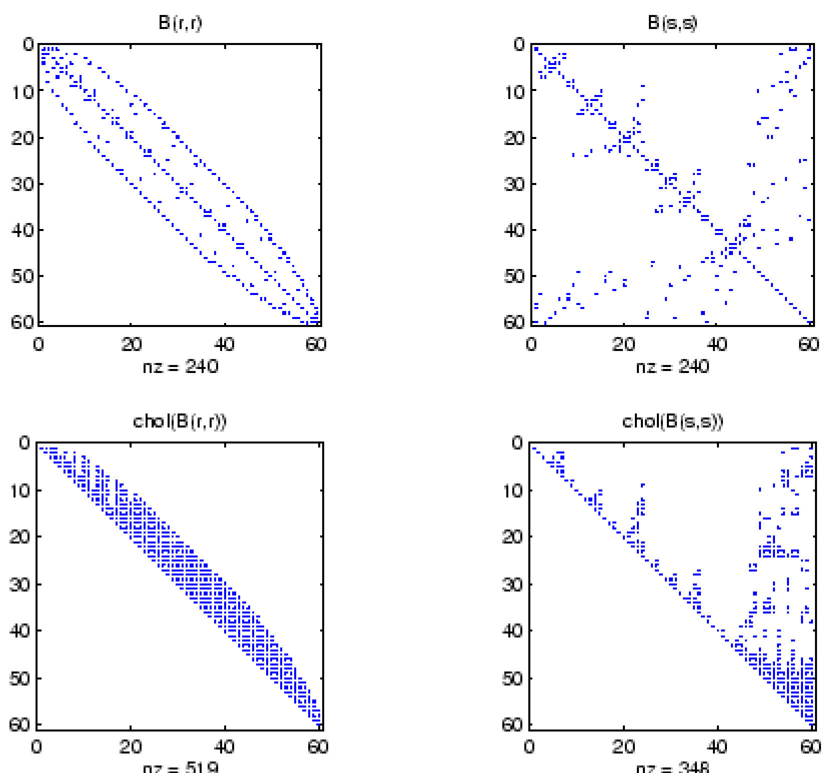
- If a row index appears two or more times in the same column, `symamd` ignores the duplicate entries, continues processing, and provides information about the duplicate entries in `stats(4:7)`.
- If row indices in a column are out of order, `symamd` sorts each column of its internal copy of the matrix `S` (but does not repair the input matrix `S`), continues processing, and provides information about the out-of-order entries in `stats(4:7)`.
- If `S` is invalid in any other way, `symamd` cannot continue. It prints an error message, and returns no output arguments (`p` or `stats`).

The ordering is followed by a symmetric elimination tree post-ordering.

Examples

Here is a comparison of reverse Cuthill-McKee and minimum degree on the Bucky ball example mentioned in the `symrcm` reference page.

```
B = bucky+4*speye(60);
r = symrcm(B);
p = symamd(B);
R = B(r,r);
S = B(p,p);
subplot(2,2,1), spy(R,4), title('B(r,r)')
subplot(2,2,2), spy(S,4), title('B(s,s)')
subplot(2,2,3), spy(chol(R),4), title('chol(B(r,r))')
subplot(2,2,4), spy(chol(S),4), title('chol(B(s,s))')
```



Even though this is a very small problem, the behavior of both orderings is typical. RCM produces a matrix with a narrow bandwidth which fills in almost completely during the Cholesky factorization. Minimum degree produces a structure with large blocks of contiguous zeros which do not fill in during the factorization. Consequently, the minimum degree ordering requires less time and storage for the factorization.

References

The authors of the code for `symamd` are Stefan I. Larimore and Timothy A. Davis (`davis@cise.ufl.edu`), University of Florida. The algorithm was developed in collaboration with John Gilbert, Xerox PARC, and Esmond Ng, Oak Ridge National Laboratory.

Sparse Matrix Algorithms Research at the University of Florida:
`http://www.cise.ufl.edu/research/sparse/`

See Also

`colamd` | `colperm` | `spparms` | `symrcm` | `amd`

Purpose

Symbolic factorization analysis

Syntax

```
count = symbfact(A)
count = symbfact(A,'sym')
count = symbfact(A,'col')
count = symbfact(A,'row')
count = symbfact(A,'lo')
[count,h,parent,post,R] = symbfact(...)
[count,h,parent,post,L] = symbfact(A,type,'lower')
```

Description

`count = symbfact(A)` returns the vector of row counts of `R=chol(A)`. `symbfact` should be much faster than `chol(A)`.

`count = symbfact(A,'sym')` is the same as `count = symbfact(A)`.

`count = symbfact(A,'col')` returns row counts of `R=chol(A'*A)` (without forming it explicitly).

`count = symbfact(A,'row')` returns row counts of `R=chol(A*A')`.

`count = symbfact(A,'lo')` is the same as `count = symbfact(A)` and uses `tril(A)`.

`[count,h,parent,post,R] = symbfact(...)` has several optional return values.

The flop count for a subsequent Cholesky factorization is `sum(count.^2)`

Return Value	Description
`h`	Height of the elimination tree
`parent`	The elimination tree itself
`post`	Postordering of the elimination tree
`R`	0-1 matrix having the structure of `chol(A)` for the symmetric case, `chol(A'*A)` for the `'col'` case, or `chol(A*A')` for the `'row'` case.

`symbfact(A)` and `symbfact(A,'sym')` use the upper triangular part of `A (triu(A))` and assume the lower triangular part is the transpose of the upper triangular part. `symbfact(A,'lo')` uses `tril(A)` instead.

`[count,h,parent,post,L] = symbfact(A,type,'lower')` where `type` is one of `'sym'`,`'col'`, `'row'`, or`'lo'` returns a lower triangular symbolic factor `L=R'`. This form is quicker and requires less memory.

See Also

`chol` | `etree` | `treelayout`

symmlq

Purpose

Symmetric LQ method

Syntax

```
x = symmlq(A,b)
symmlq(A,b,tol)
symmlq(A,b,tol,maxit)
symmlq(A,b,tol,maxit,M)
symmlq(A,b,tol,maxit,M1,M2)
symmlq(A,b,tol,maxit,M1,M2,x0)
[x,flag] = symmlq(A,b,...)
[x,flag,relres] = symmlq(A,b,...)
[x,flag,relres,iter] = symmlq(A,b,...)
[x,flag,relres,iter,resvec] = symmlq(A,b,...)
[x,flag,relres,iter,resvec,resveccg] = symmlq(A,b,...)
```

Description

`x = symmlq(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be symmetric but need not be positive definite. It should also be large and sparse. The column vector `b` must have length `n`. You can specify `A` as a function handle, `afun`, such that `afun(x)` returns `A*x`.

"Parameterizing Functions" explains how to provide additional parameters to the function `afun`, as well as the preconditioner function `mfun` described below, if necessary.

If `symmlq` converges, a message to that effect is displayed. If `symmlq` fails to converge after the maximum number of iterations or halts for any reason, a warning message is printed displaying the relative residual `norm(b-A*x)/norm(b)` and the iteration number at which the method stopped or failed.

`symmlq(A,b,tol)` specifies the tolerance of the method. If `tol` is `[]`, then `symmlq` uses the default, `1e-6`.

`symmlq(A,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]`, then `symmlq` uses the default, `min(n,20)`.

`symmlq(A,b,tol,maxit,M)` and `symmlq(A,b,tol,maxit,M1,M2)` use the symmetric positive definite preconditioner `M` or `M = M1*M2` and effectively solve the system `inv(sqrt(M))*A*inv(sqrt(M))*y =`

inv(sqrt(M))*b for y and then return x = in(sqrt(M))*y. If M is []
then symmlq applies no preconditioner. M can be a function handle mfun
such that mfun(x) returns M\x.

symmlq(A,b,tol,maxit,M1,M2,x0) specifies the initial guess. If x0 is
[], then symmlq uses the default, an all-zero vector.

[x,flag] = symmlq(A,b,...) also returns a convergence flag.

Flag	Convergence
0	symmlq converged to the desired tolerance tol within maxit iterations.
1	symmlq iterated maxit times but did not converge.
2	Preconditioner M was ill-conditioned.
3	symmlq stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during symmlq became too small or too large to continue computing.
5	Preconditioner M was not symmetric positive definite.

Whenever flag is not 0, the solution x returned is that with minimal
norm residual computed over all the iterations. No messages are
displayed if the flag output is specified.

[x,flag,relres] = symmlq(A,b,...) also returns the relative
residual norm(b-A*x)/norm(b). If flag is 0, relres <= tol.

[x,flag,relres,iter] = symmlq(A,b,...) also returns the iteration
number at which x was computed, where 0 <= iter <= maxit.

[x,flag,relres,iter,resvec] = symmlq(A,b,...) also returns a
vector of estimates of the symmlq residual norms at each iteration,
including norm(b-A*x0).

[x,flag,relres,iter,resvec,resveccg] = symmlq(A,b,...) also
returns a vector of estimates of the conjugate gradients residual norms
at each iteration.

Examples

Example 1

```
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -2*on],-1:1,n,n);
b = sum(A,2);
tol = 1e-10;
maxit = 50; M1 = spdiags(4*on,0,n,n);

x = symmlq(A,b,tol,maxit,M1);
symmlq converged at iteration 49 to a solution with relative
residual 4.3e-015
```

Example 2

This example replaces the matrix `A` in Example 1 with a handle to a matrix-vector product function `afun`. The example is contained in the function `run_symmlq` that:

- Calls `symmlq` with the function handle `@afun` as its first argument.
- Contains `afun` as a nested function, so that all variables in `run_symmlq` are available to `afun`.

The following shows the code for `run_symmlq`:

```
function x1 = run_symmlq
n = 100;
on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x1 = symmlq(@afun,b,tol,maxit,M1);

       function y = afun(x)
          y = 4 * x;
          y(2:n) = y(2:n) - 2 * x(1:n-1);
```

```
            y(1:n-1) = y(1:n-1) - 2 * x(2:n);
        end
end
```

When you enter

```
x1=run_symmlq;
```

MATLAB software displays the message

```
symmlq converged at iteration 49 to a solution with relative
residual 4.3e-015
```

Example 3

Use a symmetric indefinite matrix that fails with pcg.

```
A = diag([20:-1:1,-1:-1:-20]);
b = sum(A,2);      % The true solution is the vector of all ones.
x = pcg(A,b);      % Errors out at the first iteration.
pcg stopped at iteration 1 without converging to the desired
tolerance 1e-006 because a scalar quantity became too small or
too large to continue computing.
The iterate returned (number 0) has relative residual 1
```

However, symmlq can handle the indefinite matrix A.

```
x = symmlq(A,b,1e-6,40);
symmlq converged at iteration 39 to a solution with relative
residual 1.3e-007
```

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Paige, C. C. and M. A. Saunders, "Solution of Sparse Indefinite Systems of Linear Equations." *SIAM J. Numer. Anal.*, Vol.12, 1975, pp. 617-629.

See Also

`bicg` | `bicgstab` | `cgs` | `lsqr` | `gmres` | `minres` | `pcg` | `qmr` | `function_handle` | `mldivide`

Purpose

Sparse reverse Cuthill-McKee ordering

Syntax

```
r = symrcm(S)
```

Description

`r = symrcm(S)` returns the symmetric reverse Cuthill-McKee ordering of `S`. This is a permutation `r` such that `S(r,r)` tends to have its nonzero elements closer to the diagonal. This is a good preordering for LU or Cholesky factorization of matrices that come from long, skinny problems. The ordering works for both symmetric and nonsymmetric `S`.

For a real, symmetric sparse matrix, `S`, the eigenvalues of `S(r,r)` are the same as those of `S`, but `eig(S(r,r))` probably takes less time to compute than `eig(S)`.

Algorithms

The algorithm first finds a pseudoperipheral vertex of the graph of the matrix. It then generates a level structure by breadth-first search and orders the vertices by decreasing distance from the pseudoperipheral vertex. The implementation is based closely on the SPARSPAK implementation described by George and Liu.

Examples

The statement

```
B = bucky;
```

uses a function in the `demos` toolbox to generate the adjacency graph of a truncated icosahedron. This is better known as a soccer ball, a Buckminster Fuller geodesic dome (hence the name `bucky`), or, more recently, as a 60-atom carbon molecule. There are 60 vertices. The vertices have been ordered by numbering half of them from one hemisphere, pentagon by pentagon; then reflecting into the other hemisphere and gluing the two halves together. With this numbering, the matrix does not have a particularly narrow bandwidth, as the first spy plot shows

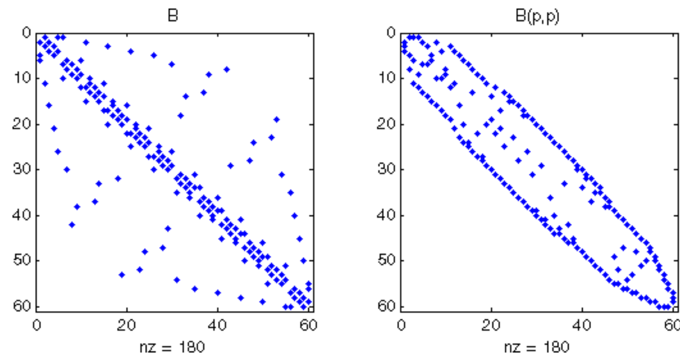
```
subplot(1,2,1),spy(B),title('B')
```

The reverse Cuthill-McKee ordering is obtained with

```
p = symrcm(B);
R = B(p,p);
```

The `spy` plot shows a much narrower bandwidth.

```
subplot(1,2,2),spy(R),title('B(p,p)')
```



This example is continued in the reference pages for `symamd`.

The bandwidth can also be computed with

```
[i,j] = find(B);
bw = max(i-j) + 1;
```

The bandwidths of `B` and `R` are 35 and 12, respectively.

References

[1] George, Alan and Joseph Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[2] Gilbert, John R., Cleve Moler, and Robert Schreiber, “Sparse Matrices in MATLAB: Design and Implementation,” *SIAM Journal on Matrix Analysis*, 1992. A slightly expanded version is also available as a technical report from the Xerox Palo Alto Research Center.

See Also

`colamd` | `colperm` | `symamd`

Purpose

Determine symbolic variables in expression

Syntax

```
symvar 'expr'
s = symvar('expr')
```

Description

`symvar 'expr'` searches the expression, `expr`, for identifiers other than `i`, `j`, `pi`, `inf`, `nan`, `eps`, and common functions. `symvar` displays those variables that it finds or, if no such variable exists, displays an empty cell array, `{}`.

`s = symvar('expr')` returns the variables in a cell array of strings, `s`. If no such variable exists, `s` is an empty cell array.

Examples

`symvar` finds variables `beta1` and `x`, but skips `pi` and the `cos` function.

```
symvar 'cos(pi*x - beta1)'

ans =

    'beta1'
    'x'
```

See Also

`strfind`

Purpose Two ways to call MATLAB functions

Description You can call MATLAB functions using either *command syntax* or *function syntax*, as described below.

Command Syntax

A function call in this syntax consists of the function name followed by one or more arguments separated by spaces:

```
functionname arg1 arg2 ... argn
```

Command syntax does not allow you to obtain any values that might be returned by the function. Attempting to assign output from the function to a variable using command syntax generates an error. Use function syntax instead.

Examples of command syntax:

```
save mydata.mat x y z
import java.awt.Button java.lang.String
```

Arguments are treated as string literals. See the examples below, under "Argument Passing" on page 1-5207.

Function Syntax

A function call in this syntax consists of the function name followed by one or more arguments separated by commas and enclosed in parentheses:

```
functionname(arg1, arg2, ..., argn)
```

You can assign the output of the function to one or more output values. When assigning to more than one output variable, separate the variables by commas or spaces and enclose them in square brackets (`[]`):

```
[out1,out2,...,outn] = functionname(arg1, arg2, ..., argn)
```

Examples of function syntax:

```
copyfile('srcfile', '..\mytests', 'writable')
[x1,x2,x3,x4] = deal(A{:})
```

Arguments are passed to the function by value. See the examples below, under “Argument Passing” on page 1-5207.

Argument Passing

When calling a function using command syntax, MATLAB passes the arguments as string literals. When using function syntax, arguments are passed by value.

In the following example, assign a value to `A` and then call `disp` on the variable to display the value passed. Calling `disp` with command syntax passes the variable name, `'A'`:

```
A = pi;
disp A
   A
```

while function syntax passes the value assigned to `A`:

```
A = pi;
disp(A)
   3.1416
```

The next example passes two strings to `strcmp` for comparison. Calling the function with command syntax compares the variable names, `'str1'` and `'str2'`:

```
str1 = 'one';    str2 = 'one';
strcmp str1 str2
ans =
     0        (unequal)
```

while function syntax compares the values assigned to the variables, `'one'` and `'one'`:

```
str1 = 'one';    str2 = 'one';
strcmp(str1, str2)
```

```
ans =
     1        (equal)
```

Passing Strings

When using the function syntax to pass a string literal to a function, you must enclose the string in single quotes, ('string'). For example, to create a new folder called myapptests, use

```
mkdir('myapptests')
```

On the other hand, variables that contain strings do not need to be enclosed in quotes:

```
folder = 'myapptests';
mkdir(folder)
```

See Also

mlint

How To

- "Check Code for Errors and Warnings"

Purpose

Execute operating system command and return output

Syntax

```
status = system(command)
[status,cmdout] = system(command)
[status,cmdout] = system(command,'-echo')
```

Description

`status = system(command)` calls the operating system to execute the specified command. The operation waits for the command to finish execution before returning the exit status of the command to the `status` variable.

`[status,cmdout] = system(command)` additionally returns the output of the command to `cmdout`. This syntax is most useful for commands that do not require user input, such as `dir`.

`[status,cmdout] = system(command,'-echo')` additionally displays (echoes) the command output in the MATLAB Command Window. This syntax is most useful for commands that require user input and that run correctly in the MATLAB Command Window.

Input Arguments

command - Operating system command

string

Operating system command, specified as a string. The `command` executes in a system shell, which might not be the shell from which you launched MATLAB.

Example: `'dir'`

Example: `'ls'`

Output Arguments

status - Command exit status

0 | nonzero integer

Command exit status, returned as either `0` or a nonzero integer. When the command is successful, `status` is `0`. Otherwise, `status` is a nonzero integer.

- If `command` includes the ampersand character (&), then `status` is the exit status upon command launch.
- If `command` does not include the ampersand character (&), then `status` is the exit status upon command completion.

cmdout - Output of operating system command
string

Output of the operating system command, returned as a string.

Examples

Display Windows Operating System Command Status and Output

On a Windows system, display the current folder using the `cd` command.

```
command = 'cd';
[status,cmdout] = system(command)
```

```
status =
     0
cmdout =
C:\matlab\myfiles
```

A `status` of zero indicates that the command completed successfully. MATLAB returns a string containing the current folder in `cmdout`.

Save UNIX Command Exit Status and Output

List all users who are currently logged in, and save the command exit status and output. Then, view the status.

```
command = 'who';
[status,cmdout] = unix(command);
status
```

```
status =

     0
```

A `status` of zero indicates that the command completed successfully. MATLAB returns a string containing the list of users in `cmdout`.

Limitations

- DOS does not support UNC path names. Therefore, if the current folder uses a UNC path name, then running `system` with a DOS `command` that relies on the current folder fails. To work around this limitation, change the folder to a mapped drive before calling `system`.

Algorithms

On UNIX, MATLAB uses a shell program to execute the given command. It determines which shell program to use by checking environment variables on your system. MATLAB first checks the `MATLAB_SHELL` variable, and if either empty or not defined, then checks `SHELL`. If `SHELL` is also empty or not defined, MATLAB uses `/bin/sh`.

Tips

- To execute the operating system command in the background, include the trailing character, `&`, in the `command` argument (for example, `'notepad &'` on a Windows platform, or `'emacs &'` on UNIX). The exit status is immediately returned to the `status` variable. This syntax is useful for console programs that require interactive user command input while they run, and that do not run correctly in the MATLAB Command Window.

  Note If `command` includes the trailing `&` character, `cmdout` is empty.

- On a UNIX system, the `system` function redirects `stdin` to the invoked command, `command`, by default. This redirection also forwards MATLAB script commands and the keyboard type-ahead buffer to the invoked command while the `system` function executes. This can lead to corrupted output when `system` does not complete execution immediately. To disable `stdin` and type-ahead redirection, include the formatted string `< /dev/null` in the call to the invoked command.

See Also

`computer` | `dos` | `perl` | `unix` | `!` (exclamation point)

Concepts

- "Running External Commands, Scripts, and Programs"

Purpose

Tangent of argument in radians

Syntax

```
Y = tan(X)
```

Description

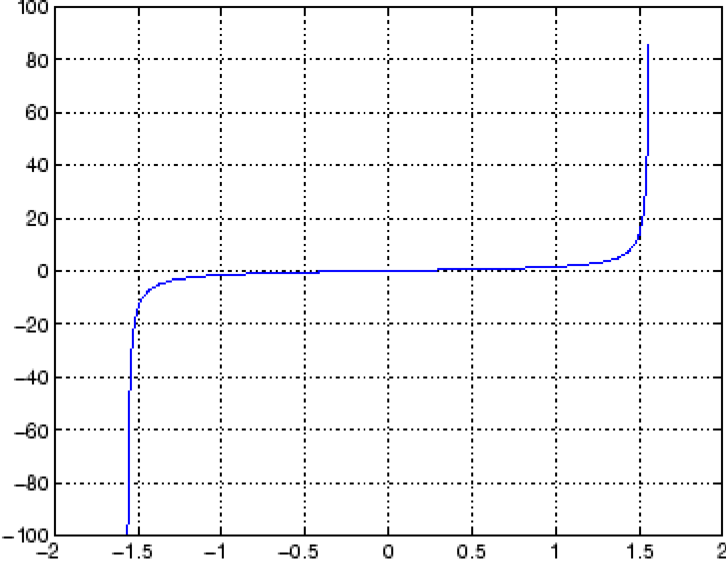
The `tan` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = tan(X)` returns the circular tangent of each element of `X`.

Examples

Graph the tangent function over the domain $-\pi/2 < x < \pi/2$.

```
x = (-pi/2)+0.01:0.01:(pi/2)-0.01;
plot(x,tan(x)), grid on
```



The expression `tan(pi/2)` does not evaluate as infinite but as the reciprocal of the floating point accuracy `eps` since `pi` is only a floating-point approximation to the exact value of π.

See Also

`tand` | `tanh` | `atan` | `atan2` | `atand` | `atan2d`

tand

Purpose Tangent of argument in degrees

Syntax

```
Y = tand(X)
```

Description

`Y = tand(X)` returns the tangent of the elements of `X`, which are expressed in degrees.

Input Arguments

X - Angle in degrees

scalar value | vector | matrix | N-D array

Angle in degrees, specified as a real-valued or complex-valued scalar, vector, matrix, or N-D array. The `tand` operation is element-wise when `X` is nonscalar.

Data Types
single | double
Complex Number Support: Yes

Output Arguments

Y - Tangent of angle

scalar value | vector | matrix | N-D array

Tangent of angle, returned as a real-valued or complex-valued scalar, vector, matrix, or N-D array of the same size as `X`.

Examples

Tangent of 90 degrees compared to tangent of /2 radians

`tand(90)` is infinite, whereas `tan(pi/2)` is large but finite.

```
tand(90)

ans =

   Inf

tan(pi/2)

ans =

   1.6331e+16
```

Tangent of vector of complex angles, specified in degrees

```
z = [180+i 15+2i 10+3i];
y = tand(z)
```

```
y =

   0 + 0.0175i   0.2676 + 0.0374i   0.1758 + 0.0539i
```

See Also

atand | tan | atan

Purpose

Hyperbolic tangent

Syntax

```
Y = tanh(X)
```

Description

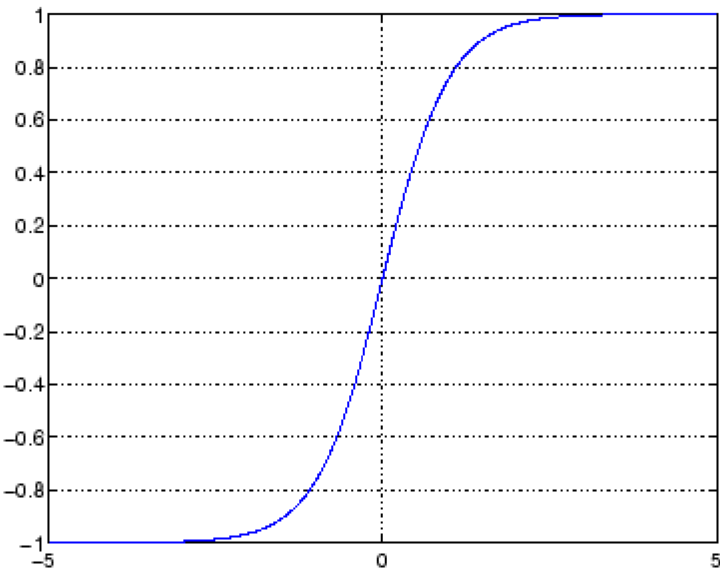
The `tanh` function operates element-wise on arrays. The function's domains and ranges include complex values. All angles are in radians.

`Y = tanh(X)` returns the hyperbolic tangent of each element of `X`.

Examples

Graph the hyperbolic tangent function over the domain $-5 \le x \le 5$.

```
x = -5:0.01:5;
plot(x,tanh(x)), grid on
```



See Also

`atan` | `atan2` | `tan` | `atanh` | `sinh` | `cosh`

Purpose

Compress files into tar file

Syntax

```
tar(tarfilename,files)
tar(tarfilename,files,rootfolder)
entrynames = tar(...)
```

Description

`tar(tarfilename,files)` creates a tar file named `tarfilename` from the list of files and folders specified in `files`. Folders recursively include all of their content. If `files` includes relative paths, the tar file also contains relative paths. The tar file does not include absolute paths.

`tar(tarfilename,files,rootfolder)` specifies the path for `files` relative to `rootfolder` rather than the current folder. Relative paths in the tar file reflect the relative paths in `files`, and do not include path information from `rootfolder`.

`entrynames = tar(...)` returns a string cell array of the names of the files contained in `tarfilename`. If `files` includes relative paths, `entrynames` also contains relative paths.

Tips

`tar` cannot compress folders larger than 2 GB.

Input Arguments

tarfilename

String specifying the name of the tar file. If `tarfilename` has no extension, MATLAB appends the `.tar` extension. The `tarfilename` extension can end in `.tgz` or `.gz`. In this case, `tarfilename` is gzipped.

files

String or cell array of strings containing the list of files or folders included in `tarfilename`.

Individual files that are on the MATLAB path can be specified as partial path names. Otherwise an individual file can be specified relative to the current folder or with an absolute path.

Folders must be specified relative to the current folder or with absolute paths. On UNIX systems, folders can also start with `~/` or `~username/`,

which expands to the current user's home folder or the specified user's home folder, respectively. The wildcard character `*` can be used when specifying files or folders, except when relying on the MATLAB path to resolve a file name or partial path name.

rootfolder

String specifying the path for *`files`*.

Examples

Tar all files in the current folder to the file `backup.tgz`.

```
tar('backup.tgz','.');
```

See Also

`gzip` | `gunzip` | `untar` | `unzip` | `zip`

Purpose

Name of system's temporary folder

Syntax

```
tmp_folder = tempdir
```

Description

`tmp_folder = tempdir` returns the name of the system's temporary folder, if one exists. This function does not create a new folder.

See Also

`delete` | `recycle` | `tempname`

How To

- "Creating Temporary Files"

tempname

Purpose Unique name for temporary file

Syntax

```
tmp_nam = tempname
```

Description `tmp_nam = tempname` returns a unique string, `tmp_nam`, suitable for use as a temporary filename.

Note The filename that `tempname` generates is not guaranteed to be unique; however, it is likely to be so.

See Also `tempdir`

How To

- “Creating Temporary Files”

Purpose

Tetrahedron mesh plot

Syntax

```
tetramesh(T,X,c)
tetramesh(T,X)
tetramesh(TR)
h = tetramesh(...)
tetramesh(...,'param','value','param','value'...)
```

Description

`tetramesh(T,X,c)` displays the tetrahedrons defined in the `m`-by-4 matrix `T` as mesh. `T` is usually the output of a Delaunay triangulation of a 3-D set of points. A row of `T` contains indices into `X` of the vertices of a tetrahedron. `X` is an `n`-by-3 matrix, representing `n` points in 3 dimension. The tetrahedron colors are defined by the vector `C`, which is used as indices into the current colormap.

`tetramesh(T,X)` uses `C = 1:m` as the color for the `m` tetrahedra. Each tetrahedron has a different color (modulo the number of colors available in the current colormap).

`tetramesh(TR)` displays the tetrahedra in a triangulation representation.

`h = tetramesh(...)` returns a vector of tetrahedron handles. Each element of `h` is a handle to the set of patches forming one tetrahedron. You can use these handles to view a particular tetrahedron by turning the patch `'Visible'` property `'on'` or `'off'`.

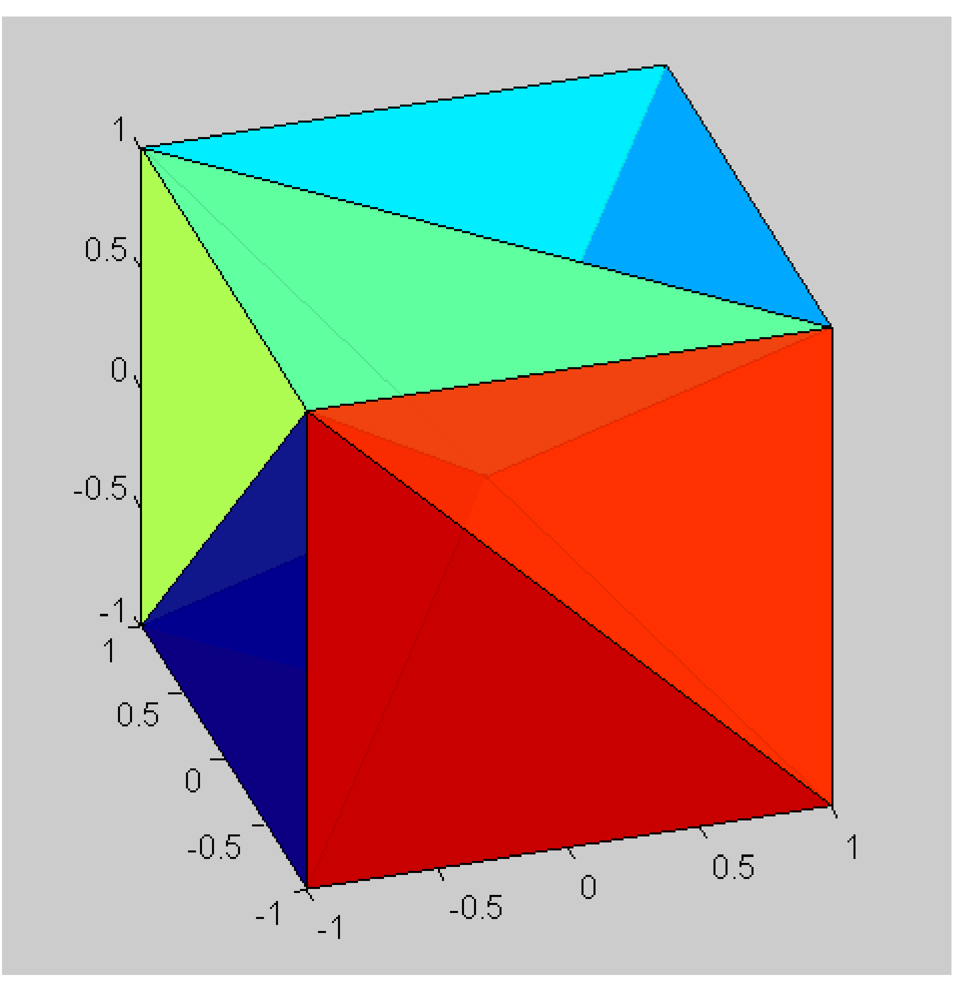
`tetramesh(...,'param','value','param','value'...)` allows additional patch property name/property value pairs to be used when displaying the tetrahedrons. For example, the default transparency parameter is set to `0.9`. You can overwrite this value by using the property name/property value pair `('FaceAlpha',value)` where `value` is a number between `0` and `1`. See `Patch Properties` for information about the available properties.

Examples

Generate a 3-D Delaunay triangulation, then use `tetramesh` to visualize the tetrahedrons.

```
d = [-1 1];
```

```
[x,y,z] = meshgrid(d,d,d); % a cube
x = [x(:);0];
y = [y(:);0];
z = [z(:);0];
% [x,y,z] are corners of a cube plus the center.
DT = delaunayTriangulation(x,y,z);
tetramesh(DT);
camorbit(20,0)
```



See Also

`trimesh` | `trisurf` | `patch` | `delaunayn` | `triangulation` | `delaunayTriangulation` | `freeBoundary(triangulation)`

matlab.unittest.Test

Superclasses

TestSuite

Purpose

Specification of a single test method

Description

The matlab.unittest.Test class holds the information needed for the TestRunner object to be able to run a single Test method of a TestCase class. A scalar Test instance is the fundamental element contained in TestSuite arrays. A simple array of Test instances is a commonly used form of a TestSuite array.

Properties

Name

Name of the Test element.

Attributes:

Dependent	true
GetAccess	public
SetAccess	immutable

Examples

Show Class of a TestSuite Array

Create a suite of Test objects of all test methods in the `BankAccountTest` class.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromClass(?BankAccountTest);

whos suite
```

```
  Name       Size             Bytes  Class                    Attributes

  suite      1x5               1636  matlab.unittest.Test
```

Each test is a matlab.unittest.Test object.

Display test method names.

```
{suite.Name}'

ans =

    'BankAccountTest/testConstructor'
    'BankAccountTest/testConstructorNotEnoughInputs'
    'BankAccountTest/testDesposit'
    'BankAccountTest/testWithdraw'
    'BankAccountTest/testNotifyInsufficientFunds'
```

See Also

`matlab.unittest.TestSuite` | `matlab.unittest.TestRunner` |

Concepts

- Property Attributes

matlab.unittest.TestCase

Purpose

Superclass of all `matlab.unittest` test classes

Description

The TestCase class is the means by which a test is written in the `matlab.unittest` framework. It provides the means to write and identify test content, as well as test fixture setup and teardown routines. Creating such a test requires deriving from TestCase to produce a TestCase subclass. Then, subclasses can leverage the metadata attributes to specify tests and test fixtures.

Construction

The matlab.unittest.TestRunner class constructs instances of the TestCase class.

Methods

addTeardown	Dynamically add teardown routine
assertClass	Assert exact class of specified value
assertEmpty	Assert value is empty
assertEqual	Assert value is equal to specified value
assertError	Assert function throws specified exception
assertFail	Produce unconditional assertion failure
assertFalse	Assert value is false
assertGreaterThan	Assert value is greater than specified value
assertGreaterThanOrEqual	Assert value is greater than or equal to specified value
assertInstanceOf	Assert value is object of specified type

assertLength	Assert value has specified length
assertLessThan	Assert value is less than specified value
assertLessThanOrEqual	Assert value is less than or equal to specified value
assertMatches	Assert string matches specified regular expression
assertNotEmpty	Assert value is not empty
assertNotEqual	Assert value is not equal to specified value
assertNotSameHandle	Assert value is not handle to specified instance
assertNumElements	Assert value has specified element count
assertReturnsTrue	Assert function returns true when evaluated
assertSameHandle	Assert two values are handles to same instance
assertSize	Assert value has specified size
assertSubstring	Assert string contains specified string
assertThat	Assert that value meets specified constraint
assertTrue	Assert value is true
assertWarning	Assert function issues specified warning
assertWarningFree	Assert function issues no warnings

assumeClass	Assume exact class of specified value
assumeEmpty	Assume value is empty
assumeEqual	Assume value is equal to specified value
assumeError	Assume function throws specified exception
assumeFail	Produce unconditional assumption failure
assumeFalse	Assume value is false
assumeGreaterThan	Assume value is greater than specified value
assumeGreaterThanOrEqual	Assume value is greater than or equal to specified value
assumeInstanceOf	Assume value is object of specified type
assumeLength	Assume value has specified length
assumeLessThan	Assume value is less than specified value
assumeLessThanOrEqual	Assume value is less than or equal to specified value
assumeMatches	Assume string matches specified regular expression
assumeNotEmpty	Assume value is not empty
assumeNotEqual	Assume value is not equal to specified value
assumeNotSameHandle	Assume value is not handle to specified instance

assumeNumElements	Assume value has specified element count
assumeReturnsTrue	Assume function returns true when evaluated
assumeSameHandle	Assume two values are handles to same instance
assumeSize	Assume value has specified size
assumeSubstring	Assume string contains specified string
assumeThat	Assume value meets specified constraint
assumeTrue	Assume value is true
assumeWarning	Assume function issues specified warning
assumeWarningFree	Assume function issues no warnings
fatalAssertClass	Fatally assert exact class of specified value
fatalAssertEmpty	Fatally assert value is empty
fatalAssertEqual	Fatally assert value is equal to specified value
fatalAssertError	Fatally assert function throws specified exception
fatalAssertFail	Produce unconditional fatal assertion failure
fatalAssertFalse	Fatally assert value is false
fatalAssertGreaterThan	Fatally assert value is greater than specified value

fatalAssertGreaterThanOrEqual	Fatally assert value is greater than or equal to specified value
fatalAssertInstanceOf	Fatally assert value is object of specified type
fatalAssertLength	Fatally assert value has specified length
fatalAssertLessThan	Fatally assert value is less than specified value
fatalAssertLessThanOrEqual	Fatally assert value is less than or equal to specified value
fatalAssertMatches	Fatally assert string matches specified regular expression
fatalAssertNotEmpty	Fatally assert value is not empty
fatalAssertNotEqual	Fatally assert value is not equal to specified value
fatalAssertNotSameHandle	Fatally assert value is not handle to specified instance
fatalAssertNumElements	Fatally assert value has specified element count
fatalAssertReturnsTrue	Fatally assert function returns true when evaluated
fatalAssertSameHandle	Fatally assert two values are handles to same instance
fatalAssertSize	Fatally assert value has specified size
fatalAssertSubstring	Fatally assert string contains specified string
fatalAssertThat	Fatally assert value meets specified constraint
fatalAssertTrue	Fatally assert value is true

fatalAssertWarning	Fatally assert function issues specified warning
fatalAssertWarningFree	Fatally assert function issues no warnings
run	Run `TestCase` test
verifyClass	Verify exact class of specified value
verifyEmpty	Verify value is empty
verifyEqual	Verify value is equal to specified value
verifyError	Verify function throws specified exception
verifyFail	Produce unconditional verification failure
verifyFalse	Verify value is false
verifyGreaterThan	Verify value is greater than specified value
verifyGreaterThanOrEqual	Verify value is greater than or equal to specified value
verifyInstanceOf	Verify value is object of specified type
verifyLength	Verify value has specified length
verifyLessThan	Verify value is less than specified value
verifyLessThanOrEqual	Verify value is less than or equal to specified value
verifyMatches	Verify string matches specified regular expression
verifyNotEmpty	Verify value is not empty

verifyNotEqual	Verify value is not equal to specified value
verifyNotSameHandle	Verify value is not handle to specified instance
verifyNumElements	Verify value has specified element count
verifyReturnsTrue	Verify function returns true when evaluated
verifySameHandle	Verify two values are handles to same instance
verifySize	Verify value has specified size
verifySubstring	Verify string contains specified string
verifyThat	Verify value meets given constraint
verifyTrue	Verify value is true
verifyWarning	Verify function issues specified warning
verifyWarningFree	Verify function issues no warnings

Attributes

Classes that derive from TestCase can define methods blocks which contain `matlab.unittest` framework-specific attributes to specify test content.

`Test`	`Method block to contain test methods.`
`TestMethodSetup`	`Method block to contain setup code.`

`TestMethodTeardown`	Method block to contain teardown code.
`TestClassSetup`	Method block to contain class level setup code.
`TestClassTeardown`	Method block to contain class level teardown code.

Examples

Create Test Case Class

Create a test case class, `FigurePropertiesTest`, with `TestMethodSetup` and `TestMethodTeardown` methods.

```
classdef FigurePropertiesTest < matlab.unittest.TestCase

    properties
        TestFigure
    end

    methods(TestMethodSetup)
        function createFigure(testCase)
            testCase.TestFigure = figure;
        end
    end

    methods(TestMethodTeardown)
        function closeFigure(testCase)
            close(testCase.TestFigure);
        end
    end

    methods(Test)

        function defaultCurrentPoint(testCase)

            cp = get(testCase.TestFigure, 'CurrentPoint');
            testCase.verifyEqual(cp, [0 0], ...
```

```
                'Default current point is incorrect')
        end

        function defaultCurrentObject(testCase)
            import matlab.unittest.constraints.IsEmpty;

            co = get(testCase.TestFigure, 'CurrentObject');
            testCase.verifyThat(co, IsEmpty, ...
                'Default current object should be empty');
        end

    end

end
```

See Also `TestRunner` |

Concepts

- "Method Attributes"

Purpose

Dynamically add teardown routine

Syntax

```
addTeardown(testCase,tearDownFcn)
addTeardown(testCase,tearDownFcn,arg1,...,argN)
```

Description

`addTeardown(testCase,tearDownFcn)` adds the `tearDownFcn` function handle that defines fixture teardown code to the `testCase` instance. The teardown code is executed in the reverse order to which it is added. This is known as LIFO (or Last-In-First-Out).

`addTeardown(testCase,tearDownFcn,arg1,...,argN)`

Input Arguments

testCase

matlab.unittest.TestCase instance

tearDownFcn

Function, specified as a function handle, that defines the fixture teardown code

arg1,...,argN

Input arguments, 1 through N (if any), required by `tearDownFcn`, specified by any type. The argument type is specified by the function argument list.

Examples

Call addTeardown in a TestMethodSetup Method

```
classdef SomeTest < matlab.unittest.TestCase

    methods(TestMethodSetup)
        function createFixture(testCase)
            p = path;
            testCase.addTeardown(@path, p);
            addpath(fullfile(pwd,'testHelpers'));
        end
    end
end
```

matlab.unittest.TestCase.assertClass

Purpose

Assert exact class of specified value

Syntax

```
assertClass(assertable,actual,className)
assertClass(assertable,actual,metaClass)
assertClass( ___ ,diagnostic)
```

Description

`assertClass(assertable,actual,className)` asserts that `actual` is a MATLAB value whose class is the class specified by `className`.

`assertClass(assertable,actual,metaClass)` asserts that `actual` is a MATLAB value whose class is the class specified by the `meta.class` instance `metaClass`. The instance must be an exact class match. See `assertInstanceOf` to assert inclusion in a class hierarchy.

`assertClass( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- The method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsOfClass;
  assertable.assertThat(actual, IsOfClass(className));
  assertable.assertThat(actual, IsOfClass(metaClass));
  ```

  There exists more functionality when using the `IsOfClass` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test.

className

Name of class, specified as a string.

metaClass

An instance of meta.class.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyClass`.

See Also

`assertThat` | `assertInstanceOf` |

matlab.unittest.TestCase.assertEmpty

Purpose

Assert value is empty

Syntax

```
assertEmpty(assertable,actual)
assertEmpty(assertable,actual,diagnostic)
```

Description

`assertEmpty(assertable,actual)` asserts that `actual` is an empty MATLAB value.

`assertEmpty(assertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  assertable.assertThat(actual, IsEmpty());
  ```

  This method is a convenience method. There exists more functionality when using the `IsEmpty` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyEmpty`.

See Also

`assertThat` | `assertNotEmpty` | `isempty`

matlab.unittest.TestCase.assertEqual

Purpose Assert value is equal to specified value

Syntax

```
assertEqual(assertable,actual,expected)
assertEqual( ___ ,Name,Value)
assertEqual( ___ ,diagnostic)
```

Description

`assertEqual(assertable,actual,expected)` asserts that `actual` is strictly equal to `expected`.

`assertEqual( ___ ,Name,Value)` asserts equality with additional options specified by one or more `Name,Value` pair arguments.

`assertEqual( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to any of the following:

```
import matlab.unittest.constraints.IsEqualTo;
assertable.assertThat(actual, IsEqualTo(expected));

import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.AbsoluteTolerance;
assertable.assertThat(actual, IsEqualTo(expected, ...
    'Within', AbsoluteTolerance(abstol)));

import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.RelativeTolerance;
assertable.assertThat(actual, IsEqualTo(expected, ...
    'Within', RelativeTolerance(reltol)));

import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.AbsoluteTolerance;
import matlab.unittest.constraints.RelativeTolerance;
assertable.assertThat(actual, IsEqualTo(expected, ...
    'Within', AbsoluteTolerance(abstol) | RelativeTolerance(reltol)));
```

There exists more functionality when using the `IsEqualTo`, `RelativeTolerance`, and `IsEqualTo` constraints directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test.

expected

Expected value.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Name-Value Pair Arguments

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

'AbsTol'

'RelTol'

matlab.unittest.TestCase.assertEqual

Examples See examples for `verifyEqual`.

See Also `assertThat` | `assertNotEqual` |

Purpose

Assert function throws specified exception

Syntax

```
assertError(assertable,actual,identifier)
assertError(assertable,actual,metaClass)
assertError( ___ ,diagnostic)
```

Description

`assertError(assertable,actual,identifier)` asserts that `actual` is a function handle that throws an exception with an error identifier that is equal to `identifier`.

`assertError(assertable,actual,metaClass)` asserts that `actual` is a function handle that throws an exception whose type is defined by the meta.class instance specified in `metaClass`. This method does not require the instance to be an exact class match, but rather it must be in the specified class hierarchy, and that hierarchy must include the MException class.

`assertError( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Throws;
  assertable.assertThat(actual, Throws(identifier));
  assertable.assertThat(actual, Throws(metaClass));
  ```

  There exists more functionality when using the `Throws` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test.

identifier

Error identifier, specified as a string.

metaClass

An instance of meta.class.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyError`.

See Also

`assertThat` | `assertWarning` | `MException` | `error`

matlab.unittest.TestCase.assertFail

Purpose

Produce unconditional assertion failure

Syntax

```
assertFail(assertable)
assertFail(assertable,diagnostic)
```

Description

`assertFail(assertable)` produces an unconditional assertion failure when encountered.

`assertFail(assertable,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `matlab.unittest.TestCase.verifyFail`.

matlab.unittest.TestCase.assertFalse

Purpose

Assert value is false

Syntax

```
assertFalse(assertable,actual)
assertFalse(assertable,actual,diagnostic)
```

Description

`assertFalse(assertable,actual)` asserts that `actual` is a scalar logical with the value of false.

`assertFalse(assertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of false. Therefore, entities such as empty arrays, false valued arrays, and zero doubles produce failures when used in this method, despite these entities exhibiting "false-like" behavior such as bypassing the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsFalse;
  assertable.assertThat(actual, IsFalse());
  ```

  There exists more functionality when using the `IsFalse` constraint directly via assertThat.
- Unlike assertTrue, this method may create a new constraint for each call. For performance critical uses, consider using assertTrue.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

For examples, see `verifyFalse`.

See Also

`assertThat` | `assertTrue` |

matlab.unittest.TestCase.assertGreaterThan

Purpose

Assert value is greater than specified value

Syntax

```
assertGreaterThan(assertable,actual,floor)
assertGreaterThan(assertable,actual,floor,diagnostic)
```

Description

`assertGreaterThan(assertable,actual,floor)` asserts that all elements of `actual` are greater than all the elements of `floor`.

`assertGreaterThan(assertable,actual,floor,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

```
import matlab.unittest.constraints.IsGreaterThan;
assertable.assertThat(actual, IsGreaterThan(floor));
```

  There exists more functionality when using the `IsGreaterThan` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test. `actual` must be the same size as `floor` unless either one is scalar, at which point scalar expansion occurs.

floor

Minimum value, exclusive.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyGreaterThan`.

See Also

`assertThat` | `assertGreaterThanOrEqual` | `assertLessThan` | `assertLessThanOrEqual` | `gt`

matlab.unittest.TestCase.assertGreaterThanOrEqual

Purpose

Assert value is greater than or equal to specified value

Syntax

```
assertGreaterThanOrEqual(assertable,actual,floor)
assertGreaterThanOrEqual(assertable,actual,floor,diagnostic)
```

Description

`assertGreaterThanOrEqual(assertable,actual,floor)` asserts that all elements of `actual` are greater than or equal to all the elements of `floor`.

`assertGreaterThanOrEqual(assertable,actual,floor,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsGreaterThanOrEqualTo;
  assertable.assertThat(actual, IsGreaterThanOrEqualTo(floor));
  ```

  There exists more functionality when using the `IsGreaterThanOrEqualTo` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test. `actual` must be the same size as `floor` unless either one is scalar, at which point scalar expansion occurs.

floor

Minimum value.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyGreaterThanOrEqual`.

See Also

`assertThat` | `assertGreaterThan` | `assertLessThanOrEqual` | `assertLessThan` | `ge`

matlab.unittest.TestCase.assertInstanceOf

Purpose

Assert value is object of specified type

Syntax

```
assertInstanceOf(assertable,actual,className)
assertInstanceOf(assertable,actual,metaClass)
assertInstanceOf( ___ ,diagnostic)
```

Description

`assertInstanceOf(assertable,actual,className)` asserts that `actual` is a MATLAB value whose class is the class specified by `className`.

`assertInstanceOf(assertable,actual,metaClass)` `actual` is a MATLAB value whose class is the class specified by the `meta.class` instance `metaClass`.

`assertInstanceOf( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  assertable.assertThat(actual, IsInstanceOf(className));
  assertable.assertThat(actual, IsInstanceOf(metaClass));
  ```

  There exists more functionality when using the `IsInstanceOf` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test.

className

Name of class, specified as a string.

metaClass

An instance of meta.class.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyInstanceOf`.

See Also

`assertThat` | `assertClass` | `isa`

matlab.unittest.TestCase.assertLength

Purpose

Assert value has specified length

Syntax

```
assertLength(assertable,actual,expectedLength)
assertLength(assertable,actual,expectedLength,diagnostic)
```

Description

`assertLength(assertable,actual,expectedLength)` that `actual` is a MATLAB array whose length is `expectedLength`.

`assertLength(assertable,actual,expectedLength,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

```
import matlab.unittest.constraints.HasLength;
assertable.assertThat(actual, HasLength(expectedLength));
```

  There exists more functionality when using the `HasLength` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test.

expectedLength

The length of an array is defined as the largest dimension of that array.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyLength`.

See Also

`assertThat` | `assertSize` | `assertNumElements` | `length`

matlab.unittest.TestCase.assertLessThan

Purpose

Assert value is less than specified value

Syntax

```
assertLessThan(assertable,actual,ceiling)
assertLessThan(assertable,actual,ceiling,diagnostic)
```

Description

`assertLessThan(assertable,actual,ceiling)` asserts that all elements of `actual` are less than all the elements of `ceiling`.

`assertLessThan(assertable,actual,ceiling,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThan;
  assertable.assertThat(actual, IsLessThan(ceiling));
  ```

  There exists more functionality when using the `IsLessThan` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test. `actual` must be the same size as `ceiling` unless either one is scalar, at which point scalar expansion occurs.

ceiling

Maximum value, exclusive.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyLessThan`.

See Also

`assertThat` | `assertLessThanOrEqual` | `assertGreaterThan` | `assertGreaterThanOrEqual` | `lt`

matlab.unittest.TestCase.assertLessThanOrEqual

Purpose

Assert value is less than or equal to specified value

Syntax

```
assertLessThanOrEqual(assertable,actual,ceiling)
assertLessThanOrEqual(assertable,actual,ceiling,diagnostic)
```

Description

`assertLessThanOrEqual(assertable,actual,ceiling)` asserts that all elements of `actual` are less than or equal to all the elements of `ceiling`.

`assertLessThanOrEqual(assertable,actual,ceiling,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThanOrEqualTo;
  assertable.assertThat(actual, IsLessThanOrEqualTo(ceiling));
  ```

  There exists more functionality when using the `IsLessThanOrEqualTo` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test. `actual` must be the same size as `ceiling` unless either one is scalar, at which point scalar expansion occurs.

ceiling

Maximum value.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyLessThanOrEqual`.

See Also

`assertThat` | `assertLessThan` | `assertGreaterThan` | `assertGreaterThanOrEqual` | `le`

matlab.unittest.TestCase.assertMatches

Purpose

Assert string matches specified regular expression

Syntax

```
assertMatches(assertable,actual,expression)
assertMatches(assertable,actual,expression,diagnostic)
```

Description

`assertMatches(assertable,actual,expression)` asserts that `actual` is a string that matches the regular expression defined by `expression`.

`assertMatches(assertable,actual,expression,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Matches;
  assertable.assertThat(actual, Matches(expression));
  ```

  There exists more functionality when using the `Matches` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test.

expression

The value to match, specified as a regular expression.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyMatches`.

See Also

`assertThat` | `assertSubstring` | `regexp`

matlab.unittest.TestCase.assertNotEmpty

Purpose

Assert value is not empty

Syntax

```
assertNotEmpty(assertable,actual)
assertNotEmpty(assertable,actual,diagnostic)
```

Description

`assertNotEmpty(assertable,actual)` asserts that `actual` is a non-empty MATLAB value.

`assertNotEmpty(assertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  assertable.assertThat(actual, ~IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyNotEmpty`.

See Also

`assertThat` | `assertEmpty` | `isempty`

matlab.unittest.TestCase.assertNotEqual

Purpose

Assert value is not equal to specified value

Syntax

```
assertNotEqual(assertable,actual,notExpected)
assertNotEqual(assertable,actual,notExpected,diagnostic)
```

Description

`assertNotEqual(assertable,actual,notExpected)` asserts that `actual` is not equal to `notExpected`.

`assertNotEqual(assertable,actual,notExpected,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEqualTo;
  assertable.assertThat(actual, ~IsEqualTo(notExpected));
  ```

  There exists more functionality when using the `IsEqualTo` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test.

notExpected

Value to compare.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyNotEqual`.

See Also

`assertThat` | `assertEqual` |

matlab.unittest.TestCase.assertNotSameHandle

Purpose

Assert value is not handle to specified instance

Syntax

```
assertNotSameHandle(assertable,actual,notExpectedHandle)
assertNotSameHandle(assertable,actual,notExpectedHandle,
    diagnostic)
```

Description

`assertNotSameHandle(assertable,actual,notExpectedHandle)` asserts that `actual` is a different size and/or does not contain the same instances as the `notExpectedHandle` handle array.

`assertNotSameHandle(assertable,actual,notExpectedHandle,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  assertable.assertThat(actual, ~IsSameHandleAs(notExpectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test.

notExpectedHandle

The handle array to compare.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyNotSameHandle`.

See Also

`assertThat` | `assertSameHandle` |

matlab.unittest.TestCase.assertNumElements

Purpose

Assert value has specified element count

Syntax

```
assertNumElements(assertable,actual,expectedElementCount)
assertNumElements(assertable,actual,expectedElementCount,
   diagnostic)
```

Description

`assertNumElements(assertable,actual,expectedElementCount)` asserts that `actual` is a MATLAB array with `expectedElementCount` number of elements.

`assertNumElements(assertable,actual,expectedElementCount,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

```
import matlab.unittest.constraints.HasElementCount;
assertable.assertThat(actual, HasElementCount(expectedElementCount));
```

  There exists more functionality when using the `HasElementCount` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test.

expectedElementCount

The expected number of elements in the array.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyNumElements`.

See Also

`assertThat` | `assertSize` | `assertLength` | `numel`

matlab.unittest.TestCase.assertReturnsTrue

Purpose

Assert function returns true when evaluated

Syntax

```
assertReturnsTrue(assertable,actual)
assertReturnsTrue(assertable,actual,diagnostic)
```

Description

`assertReturnsTrue(assertable,actual)` asserts that `actual` is a function handle that returns a scalar logical whose value is true.

`assertReturnsTrue(assertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- It is a shortcut for quick custom comparison functionality that can be defined quickly, and possibly inline. It can be preferable over simply evaluating the function directly and using assertTrue because the function handle will be shown in the diagnostics, thus providing more insight into the failure condition which is lost when using assertTrue.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ReturnsTrue;
  assertable.assertThat(actual, ReturnsTrue());
  ```

  There exists more functionality when using the `ReturnsTrue` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The function handle to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyReturnsTrue`.

See Also

`assertThat` | `assertTrue` |

matlab.unittest.TestCase.assertSameHandle

Purpose

Assert two values are handles to same instance

Syntax

```
assertSameHandle(assertable,actual,expectedHandle)
assertSameHandle(assertable,actual,expectedHandle,diagnostic)
```

Description

`assertSameHandle(assertable,actual,expectedHandle)` asserts that `actual` is the same size and contains the same instances as the `expectedHandle` handle array.

`assertSameHandle(assertable,actual,expectedHandle,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  assertable.assertThat(actual, IsSameHandleAs(expectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test.

expectedHandle

The expected handle array.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifySameHandle`.

See Also

`assertThat` | `assertNotSameHandle` | `handle`

matlab.unittest.TestCase.assertSize

Purpose

Assert value has specified size

Syntax

```
assertSize(assertable,actual,expectedSize)
assertSize(assertable,actual,expectedSize,diagnostic)
```

Description

`assertSize(assertable,actual,expectedSize)` asserts that `actual` is a MATLAB array whose size is `expectedSize`.

`assertSize(assertable,actual,expectedSize,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

```
import matlab.unittest.constraints.HasSize;
assertable.assertThat(actual, HasSize(expectedSize));
```

  There exists more functionality when using the `HasSize` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test.

expectedSize

The expected sizes of each dimension the array.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifySize`.

See Also

`assertThat` | `assertLength` | `assertNumElements` | `size`

matlab.unittest.TestCase.assertSubstring

Purpose

Assert string contains specified string

Syntax

```
assertSubstring(assertable,actual,substring)
assertSubstring(assertable,actual,substring,diagnostic)
```

Description

`assertSubstring(assertable,actual,substring)` asserts that `actual` is a string that contains `substring`.

`assertSubstring(assertable,actual,substring,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ContainsSubstring;
  assertable.assertThat(actual, ContainsSubstring(substring));
  ```

  There exists more functionality when using the `ContainsSubstring` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test.

substring

The value to match, specified as a string.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifySubstring`.

See Also

`assertThat` | `assertMatches` | `strfind`

matlab.unittest.TestCase.assertThat

Purpose

Assert that value meets specified constraint

Syntax

```
assertThat(assertable,actual,constraint)
assertThat(assertable,actual,constraint,diagnostic)
```

Description

`assertThat(assertable,actual,constraint)` asserts that `actual` is a value that satisfies the `constraint` provided.

If the constraint is not satisfied, an assertion failure is produced utilizing only the framework diagnostic generated by the `constraint`.

`assertThat(assertable,actual,constraint,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

When using this signature, both the diagnostic information contained within `diagnostic` is used in addition to the diagnostic information provided by the `constraint`.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test.

constraint

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyThat`.

matlab.unittest.TestCase.assertTrue

Purpose Assert value is true

Syntax

```
assertTrue(assertable,actual)
assertTrue(assertable,actual,diagnostic)
```

Description

`assertTrue(assertable,actual)` asserts that `actual` is a scalar logical with the value of true.

`assertTrue(assertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of true. Therefore, entities such as true valued arrays and non-zero doubles produce qualification failures when used in this method, despite these entities exhibiting "true-like" behavior such as triggering the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsTrue;
  assertable.assertThat(actual, IsTrue());
  ```

  There exists more functionality when using the `IsTrue` constraint directly via assertThat.

  Use of this method for performance benefits can come at the expense of less diagnostic information, and may not provide the same level of strictness adhered to by other constraints such as IsEqualTo. A similar approach that is generally less performant but can provide slightly better diagnostic information is the use of assertReturnsTrue, which at least shows the display of the function evaluated to generate the failing result.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The value to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyTrue`.

See Also

`assertThat` | `assertFalse` | `assertReturnsTrue` |

matlab.unittest.TestCase.assertWarning

Purpose

Assert function issues specified warning

Syntax

```
assertWarning(assertable,actual,warningID)
assertWarning(assertable,actual,warningID,diagnostic)
[output1,...,outputN] = assertWarning( ___ )
```

Description

`assertWarning(assertable,actual,warningID)` asserts that `actual` issues a warning with the identifier `warningID`.

`assertWarning(assertable,actual,warningID,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`[output1,...,outputN] = assertWarning( ___ )` also returns the output arguments `output1,...,outputN` that are produced when invoking `actual`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesWarnings;
  assertable.assertThat(actual, IssuesWarnings({warningID}));
  ```

  There exists more functionality when using the `IssuesWarnings` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The function handle to test.

warningID

Warning ID, specified as a string.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Output Arguments

output1,...,outputN

Output arguments, 1 through n (if any), from `actual`, returned as any type. The argument type is specified by the `actual` argument list.

Examples

See examples for `verifyWarning`.

See Also

`assertThat` | `assertError` | `assertWarningFree` | `warning`

matlab.unittest.TestCase.assertWarningFree

Purpose

Assert function issues no warnings

Syntax

```
assertWarningFree(assertable,actual)
assertWarningFree(assertable,actual,diagnostic)
[output1,...,outputN] = assertWarningFree( ___ )
```

Description

`assertWarningFree(assertable,actual)` asserts that `actual` is a function handle that issues no warnings.

`assertWarningFree(assertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`[output1,...,outputN] = assertWarningFree( ___ )` also returns the output arguments `output1,...,outputN` that are produced when invoking `actual`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesNoWarnings;
  assertable.assertThat(actual, IssuesNoWarnings());
  ```

  There exists more functionality when using the `IssuesNoWarnings` constraint directly via assertThat.

Input Arguments

assertable

The matlab.unittest.TestCase instance which is used to pass or fail the assertion in conjunction with the test running framework.

actual

The function handle to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Output Arguments

output1,...,outputN

Output arguments, 1 through n (if any), from `actual`, returned as any type. The argument type is specified by the `actual` argument list.

Examples

See examples for `verifyWarningFree`.

See Also

`assertThat` | `assertWarning` | `warning`

matlab.unittest.TestCase.assumeClass

Purpose

Assume exact class of specified value

Syntax

```
assumeClass(assumable,actual,className)
assumeClass(assumable,actual,metaClass)
assumeClass(___,diagnostic)
```

Description

`assumeClass(assumable,actual,className)` assumes that `actual` is a MATLAB value whose class is the class specified by `className`.

`assumeClass(assumable,actual,metaClass)` assumes that `actual` is a MATLAB value whose class is the class specified by the `meta.class` instance `metaClass`. The instance must be an exact class match. See assumeInstanceOf to assume inclusion in a class hierarchy.

`assumeClass(___,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- The method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsOfClass;
  assumable.assumeThat(actual, IsOfClass(className));
  assumable.assumeThat(actual, IsOfClass(metaClass));
  ```

  There exists more functionality when using the `IsOfClass` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test.

className

Name of class, specified as a string.

metaClass

An instance of meta.class.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyClass`.

See Also

`assumeThat` | `assumeInstanceOf` |

matlab.unittest.TestCase.assumeEmpty

Purpose

Assume value is empty

Syntax

```
assumeEmpty(assumable,actual)
assumeEmpty(assumable,actual,diagnostic)
```

Description

`assumeEmpty(assumable,actual)` assumes that `actual` is an empty MATLAB value.

`assumeEmpty(assumable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  assumable.assumeThat(actual, IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyEmpty`.

See Also

`assumeThat` | `assumeNotEmpty` | `isempty`

matlab.unittest.TestCase.assumeEqual

Purpose Assume value is equal to specified value

Syntax

```
assumeEqual(assumable,actual,expected)
assumeEqual( ___ ,Name,Value)
assumeEqual( ___ ,diagnostic)
```

Description

`assumeEqual(assumable,actual,expected)` assumes that `actual` is strictly equal to `expected`.

`assumeEqual( ___ ,Name,Value)` assumes equality with additional options specified by one or more `Name,Value` pair arguments.

`assumeEqual( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to any of the following:

```
import matlab.unittest.constraints.IsEqualTo;
assumable.assumeThat(actual, IsEqualTo(expected));

import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.AbsoluteTolerance;
assumable.assumeThat(actual, IsEqualTo(expected, ...
    'Within', AbsoluteTolerance(abstol)));

import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.RelativeTolerance;
assumable.assumeThat(actual, IsEqualTo(expected, ...
    'Within', RelativeTolerance(reltol)));

import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.AbsoluteTolerance;
import matlab.unittest.constraints.RelativeTolerance;
assumable.assumeThat(actual, IsEqualTo(expected, ...
    'Within', AbsoluteTolerance(abstol) | RelativeTolerance(reltol)));
```

There exists more functionality when using the `IsEqualTo`, `RelativeTolerance`, and `IsEqualTo` constraints directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test.

expected

Expected value.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Name-Value Pair Arguments

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

'AbsTol'

'RelTol'

Examples

See examples for `verifyEqual`.

See Also

`assumeThat` | `assumeNotEqual` |

Purpose

Assume function throws specified exception

Syntax

```
assumeError(assumable,actual,identifier)
assumeError(assumable,actual,metaClass)
assumeError( ___ ,diagnostic)
```

Description

`assumeError(assumable,actual,identifier)` assumes that `actual` is a function handle that throws an exception with an error identifier that is equal to `identifier`.

`assumeError(assumable,actual,metaClass)` assumes that `actual` is a function handle that throws an exception whose type is defined by the meta.class instance specified in `metaClass`. This method does not require the instance to be an exact class match, but rather it must be in the specified class hierarchy, and that hierarchy must include the MException class.

`assumeError( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Throws;
  assumable.assumeThat(actual, Throws(identifier));
  assumable.assumeThat(actual, Throws(metaClass));
  ```

  There exists more functionality when using the `Throws` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test.

identifier

Error identifier, specified as a string.

metaClass

An instance of meta.class.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyError`.

See Also

`assumeThat` | `assumeWarning` | `MException` | `error`

Purpose

Produce unconditional assumption failure

Syntax

```
assumeFail(assumable)
assumeFail(assumable,diagnostic)
```

Description

`assumeFail(assumable)` produces an unconditional assumption failure when encountered.

`assumeFail(assumable,diagnostic)`also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `matlab.unittest.TestCase.verifyFail`.

matlab.unittest.TestCase.assumeFalse

Purpose

Assume value is false

Syntax

```
assumeFalse(assumable,actual)
assumeFalse(assumable,actual,diagnostic)
```

Description

`assumeFalse(assumable,actual)` assumes that `actual` is a scalar logical with the value of false.

`assumeFalse(assumable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of false. Therefore, entities such as empty arrays, false valued arrays, and zero doubles produce failures when used in this method, despite these entities exhibiting "false-like" behavior such as bypassing the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsFalse;
  assumable.assumeThat(actual, IsFalse());
  ```

  There exists more functionality when using the `IsFalse` constraint directly via assumeThat.
- Unlike assumeTrue, this method may create a new constraint for each call. For performance critical uses, consider using assumeTrue.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

For examples, see `verifyFalse`.

See Also

`assumeThat` | `assumeTrue` |

matlab.unittest.TestCase.assumeGreaterThan

Purpose

Assume value is greater than specified value

Syntax

```
assumeGreaterThan(assumable,actual,floor)
assumeGreaterThan(assumable,actual,floor,diagnostic)
```

Description

`assumeGreaterThan(assumable,actual,floor)` assumes that all elements of `actual` are greater than all the elements of `floor`.

`assumeGreaterThan(assumable,actual,floor,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsGreaterThan;
  assumable.assumeThat(actual, IsGreaterThan(floor));
  ```

  There exists more functionality when using the `IsGreaterThan` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test. `actual` must be the same size as `floor` unless either one is scalar, at which point scalar expansion occurs.

floor

Minimum value, exclusive.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyGreaterThan`.

See Also

`assumeThat` | `assumeGreaterThanOrEqual` | `assumeLessThanOrEqual` | `assumeLessThan` | `gt`

matlab.unittest.TestCase.assumeGreaterThanOrEqual

Purpose

Assume value is greater than or equal to specified value

Syntax

```
assumeGreaterThanOrEqual(assumable,actual,floor)
assumeGreaterThanOrEqual(assumable,actual,floor,diagnostic)
```

Description

`assumeGreaterThanOrEqual(assumable,actual,floor)` assumes that all elements of `actual` are greater than or equal to all the elements of `floor`.

`assumeGreaterThanOrEqual(assumable,actual,floor,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsGreaterThanOrEqualTo;
  assumable.assumeThat(actual, IsGreaterThanOrEqualTo(floor));
  ```

  There exists more functionality when using the `IsGreaterThanOrEqualTo` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test. `actual` must be the same size as `floor` unless either one is scalar, at which point scalar expansion occurs.

floor

Minimum value.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyGreaterThanOrEqual`.

See Also

`assumeThat` | `assumeGreaterThan` | `assumeLessThanOrEqual` | `assumeLessThan` | `ge`

matlab.unittest.TestCase.assumeInstanceOf

Purpose

Assume value is object of specified type

Syntax

```
assumeInstanceOf(assumable,actual,className)
assumeInstanceOf(assumable,actual,metaClass)
assumeInstanceOf( ___ ,diagnostic)
```

Description

`assumeInstanceOf(assumable,actual,className)` assumes that `actual` is a MATLAB value whose class is the class specified by `className`.

`assumeInstanceOf(assumable,actual,metaClass)` `actual` is a MATLAB value whose class is the class specified by the `meta.class` instance `metaClass`.

`assumeInstanceOf( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsInstanceOf;
  assumable.assumeThat(actual, IsInstanceOf(className));
  assumable.assumeThat(actual, IsInstanceOf(metaClass));
  ```

  There exists more functionality when using the `IsInstanceOf` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test.

className

Name of class, specified as a string.

metaClass

An instance of meta.class.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyInstanceOf`.

See Also

`assumeThat` | `assumeClass` | `isa`

matlab.unittest.TestCase.assumeLength

Purpose

Assume value has specified length

Syntax

```
assumeLength(assumable,actual,expectedLength)
assumeLength(assumable,actual,expectedLength,diagnostic)
```

Description

`assumeLength(assumable,actual,expectedLength)` assumes that `actual` is a MATLAB array whose length is `expectedLength`.

`assumeLength(assumable,actual,expectedLength,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasLength;
  assumable.assumeThat(actual, HasLength(expectedLength));
  ```

  There exists more functionality when using the `HasLength` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test.

expectedLength

The length of an array is defined as the largest dimension of that array.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyLength`.

See Also

`assumeThat` | `assumeSize` | `assumeNumElements` | `length`

matlab.unittest.TestCase.assumeLessThan

Purpose

Assume value is less than specified value

Syntax

```
assumeLessThan(assumable,actual,ceiling)
assumeLessThan(assumable,actual,ceiling,diagnostic)
```

Description

`assumeLessThan(assumable,actual,ceiling)` assumes that all elements of `actual` are less than all the elements of `ceiling`.

`assumeLessThan(assumable,actual,ceiling,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThan;
  assumable.assumeThat(actual, IsLessThan(ceiling));
  ```

  There exists more functionality when using the `IsLessThan` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test. `actual` must be the same size as `ceiling` unless either one is scalar, at which point scalar expansion occurs.

ceiling

Maximum value, exclusive.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyLessThan`.

See Also

`assumeThat` | `assumeLessThanOrEqual` | `assumeGreaterThan` | `assumeGreaterThanOrEqual` | `lt`

matlab.unittest.TestCase.assumeLessThanOrEqual

Purpose

Assume value is less than or equal to specified value

Syntax

```
assumeLessThanOrEqual(assumable,actual,ceiling)
assumeLessThanOrEqual(assumable,actual,ceiling,diagnostic)
```

Description

`assumeLessThanOrEqual(assumable,actual,ceiling)` assumes that all elements of `actual` are less than or equal to all the elements of `ceiling`.

`assumeLessThanOrEqual(assumable,actual,ceiling,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThanOrEqualTo;
  assumable.assumeThat(actual, IsLessThanOrEqualTo(ceiling));
  ```

  There exists more functionality when using the `IsLessThanOrEqualTo` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test. `actual` must be the same size as `ceiling` unless either one is scalar, at which point scalar expansion occurs.

ceiling

Maximum value.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyLessThanOrEqual`.

See Also

`assumeThat` | `assumeLessThan` | `assumeGreaterThan` | `assumeGreaterThanOrEqual` | `le`

matlab.unittest.TestCase.assumeMatches

Purpose

Assume string matches specified regular expression

Syntax

```
assumeMatches(assumable,actual,expression)
assumeMatches(assumable,actual,expression,diagnostic)
```

Description

`assumeMatches(assumable,actual,expression)` assumes that `actual` is a string that matches the regular expression defined by `expression`.

`assumeMatches(assumable,actual,expression,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Matches;
  assumable.assumeThat(actual, Matches(expression));
  ```

  There exists more functionality when using the `Matches` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The string to test.

expression

The value to match, specified as a regular expression.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyMatches`.

See Also

`assumeThat` | `assumeSubstring` | `regexp`

matlab.unittest.TestCase.assumeNotEmpty

Purpose

Assume value is not empty

Syntax

```
assumeNotEmpty(assumable,actual)
assumeNotEmpty(assumable,actual,diagnostic)
```

Description

`assumeNotEmpty(assumable,actual)` assumes that `actual` is a non-empty MATLAB value.

`assumeNotEmpty(assumable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  assumable.assumeThat(actual, ~IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyNotEmpty`.

See Also

`assumeThat` | `assumeEmpty` | `isempty`

matlab.unittest.TestCase.assumeNotEqual

Purpose

Assume value is not equal to specified value

Syntax

```
assumeNotEqual(assumable,actual,notExpected)
assumeNotEqual(assumable,actual,notExpected,diagnostic)
```

Description

`assumeNotEqual(assumable,actual,notExpected)` assumes that `actual` is not equal to `notExpected`.

`assumeNotEqual(assumable,actual,notExpected,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEqualTo;
  assumable.assumeThat(actual, ~IsEqualTo(notExpected));
  ```

  There exists more functionality when using the `IsEqualTo` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test.

notExpected

Value to compare.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyNotEqual`.

See Also

`assumeThat` | `assumeEqual` |

matlab.unittest.TestCase.assumeNotSameHandle

Purpose

Assume value is not handle to specified instance

Syntax

```
assumeNotSameHandle(assumable,actual,notExpectedHandle)
assumeNotSameHandle(assumable,actual,notExpectedHandle,
   diagnostic)
```

Description

`assumeNotSameHandle(assumable,actual,notExpectedHandle)` assumes that `actual` is a different size and/or does not contain the same instances as the `notExpectedHandle` handle array.

`assumeNotSameHandle(assumable,actual,notExpectedHandle,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  assumable.assumeThat(actual, ~IsSameHandleAs(notExpectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test.

notExpectedHandle

The handle array to compare.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyNotSameHandle`.

See Also

`assumeThat` | `assumeSameHandle` |

matlab.unittest.TestCase.assumeNumElements

Purpose

Assume value has specified element count

Syntax

```
assumeNumElements(assumable,actual,expectedElementCount)
assumeNumElements(assumable,actual,expectedElementCount,
   diagnostic)
```

Description

`assumeNumElements(assumable,actual,expectedElementCount)` assumes that `actual` is a MATLAB array with `expectedElementCount` number of elements.

`assumeNumElements(assumable,actual,expectedElementCount,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasElementCount;
  assumable.assumeThat(actual, HasElementCount(expectedElementCount));
  ```

  There exists more functionality when using the `HasElementCount` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test.

expectedElementCount

The expected number of elements in the array.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyNumElements`.

See Also

`assumeThat` | `assumeSize` | `assumeLength` | `numel`

matlab.unittest.TestCase.assumeReturnsTrue

Purpose

Assume function returns true when evaluated

Syntax

```
assumeReturnsTrue(assumable,actual)
assumeReturnsTrue(assumable,actual,diagnostic)
```

Description

`assumeReturnsTrue(assumable,actual)` assumes that `actual` is a function handle that returns a scalar logical whose value is true.

`assumeReturnsTrue(assumable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- It is a shortcut for quick custom comparison functionality that can be defined quickly, and possibly inline. It can be preferable over simply evaluating the function directly and using assumeTrue because the function handle will be shown in the diagnostics, thus providing more insight into the failure condition which is lost when using assumeTrue.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ReturnsTrue;
  assumable.assumeThat(actual, ReturnsTrue());
  ```

  There exists more functionality when using the `ReturnsTrue` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The function handle to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyReturnsTrue`.

See Also

`assumeThat` | `assumeTrue` |

matlab.unittest.TestCase.assumeSameHandle

Purpose

Assume two values are handles to same instance

Syntax

```
assumeSameHandle(assumable,actual,expectedHandle)
assumeSameHandle(assumable,actual,expectedHandle,diagnostic)
```

Description

`assumeSameHandle(assumable,actual,expectedHandle)` assumes that `actual` is the same size and contains the same instances as the `expectedHandle` handle array.

`assumeSameHandle(assumable,actual,expectedHandle,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  assumable.assumeThat(actual, IsSameHandleAs(expectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test.

expectedHandle

The expected handle array.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifySameHandle`.

See Also

`assumeThat` | `assumeNotSameHandle` | `handle`

matlab.unittest.TestCase.assumeSize

Purpose

Assume value has specified size

Syntax

```
assumeSize(assumable,actual,expectedSize)
assumeSize(assumable,actual,expectedSize,diagnostic)
```

Description

`assumeSize(assumable,actual,expectedSize)` assumes that `actual` is a MATLAB array whose size is `expectedSize`.

`assumeSize(assumable,actual,expectedSize,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

```
import matlab.unittest.constraints.HasSize;
assumable.assumeThat(actual, HasSize(expectedSize));
```

  There exists more functionality when using the `HasSize` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test.

expectedSize

The expected sizes of each dimension the array.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifySize`.

See Also

`assumeThat` | `assumeLength` | `assumeNumElements` | `size`

matlab.unittest.TestCase.assumeSubstring

Purpose

Assume string contains specified string

Syntax

```
assumeSubstring(assumable,actual,substring)
assumeSubstring(assumable,actual,substring,diagnostic)
```

Description

`assumeSubstring(assumable,actual,substring)` assumes that `actual` is a string that contains `substring`.

`assumeSubstring(assumable,actual,substring,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

```
import matlab.unittest.constraints.ContainsSubstring;
assumable.assumeThat(actual, ContainsSubstring(substring));
```

  There exists more functionality when using the `ContainsSubstring` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test.

substring

The value to match, specified as a string.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifySubstring`.

See Also

`assumeThat` | `assumeMatches` | `strfind`

matlab.unittest.TestCase.assumeThat

Purpose

Assume value meets specified constraint

Syntax

```
assumeThat(assumable,actual,constraint)
assumeThat(assumable,actual,constraint,diagnostic)
```

Description

`assumeThat(assumable,actual,constraint)` assumes that `actual` is a value that satisfies the `constraint` provided.

If the constraint is not satisfied, an assumption failure is produced utilizing only the framework diagnostic generated by the `constraint`.

`assumeThat(assumable,actual,constraint,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

When using this signature, both the diagnostic information contained within `diagnostic` is used in addition to the diagnostic information provided by the `constraint`.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test.

constraint

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyThat`.

matlab.unittest.TestCase.assumeTrue

Purpose

Assume value is true

Syntax

```
assumeTrue(assumable,actual)
assumeTrue(assumable,actual,diagnostic)
```

Description

`assumeTrue(assumable,actual)` assumes that `actual` is a scalar logical with the value of true.

`assumeTrue(assumable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of true. Therefore, entities such as true valued arrays and nonzero doubles produce qualification failures when used in this method, despite these entities exhibiting "true-like" behavior such as triggering the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsTrue;
  assumable.assumeThat(actual, IsTrue());
  ```

  There exists more functionality when using the `IsTrue` constraint directly via assumeThat.

  Use of this method for performance benefits can come at the expense of less diagnostic information, and may not provide the same level of strictness adhered to by other constraints such as IsEqualTo. A similar approach that is generally less performant but can provide slightly better diagnostic information is the use of assumeReturnsTrue, which at least shows the display of the function evaluated to generate the failing result.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The value to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyTrue`.

See Also

`assumeThat` | `assumeFalse` | `assumeReturnsTrue` |

matlab.unittest.TestCase.assumeWarning

Purpose

Assume function issues specified warning

Syntax

```
assumeWarning(assumable,actual,warningID)
assumeWarning(assumable,actual,warningID,diagnostic)
[output1,...,outputN] = assumeWarning( ___ )
```

Description

`assumeWarning(assumable,actual,warningID)` assumes that `actual` issues a warning with the identifier `warningID`.

`assumeWarning(assumable,actual,warningID,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`[output1,...,outputN] = assumeWarning( ___ )` also return the output arguments `output1,...,outputN` that are produced when invoking `actual`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesWarnings;
  assumable.assumeThat(actual, IssuesWarnings({warningID}));
  ```

  There exists more functionality when using the `IssuesWarnings` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The function handle to test.

warningID

Warning ID, specified as a string.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Output Arguments

output1,...,outputN

Output arguments, 1 through n (if any), from `actual`, returned as any type. The argument type is specified by the `actual` argument list.

Examples

See examples for `verifyWarning`.

See Also

`assumeThat` | `assumeError` | `assumeWarningFree` | `warning`

matlab.unittest.TestCase.assumeWarningFree

Purpose

Assume function issues no warnings

Syntax

```
assumeWarningFree(assumable,actual)
assumeWarningFree(assumable,actual,diagnostic)
[output1,...,outputN] = assumeWarningFree( ___ )
```

Description

`assumeWarningFree(assumable,actual)` assumes that `actual` is a function handle that issues no warnings.

`assumeWarningFree(assumable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`[output1,...,outputN] = assumeWarningFree( ___ )` also returns the output arguments `output1,...,outputN` that are produced when invoking `actual`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesNoWarnings;
  assumable.assumeThat(actual, IssuesNoWarnings());
  ```

  There exists more functionality when using the `IssuesNoWarnings` constraint directly via assumeThat.

Input Arguments

assumable

The matlab.unittest.TestCase instance which is used to pass or fail the assumption in conjunction with the test running framework.

actual

The function handle to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Output Arguments

output1,...,outputN

Output arguments, 1 through n (if any), from `actual`, returned as any type. The argument type is specified by the `actual` argument list.

Examples

See examples for `verifyWarningFree`.

See Also

`assumeThat` | `assumeWarning` | `warning`

matlab.unittest.TestCase.fatalAssertClass

Purpose

Fatally assert exact class of specified value

Syntax

```
fatalAssertClass(fatalAssertable,actual,className)
fatalAssertClass(fatalAssertable,actual,metaClass)
fatalAssertClass( ___ ,diagnostic)
```

Description

`fatalAssertClass(fatalAssertable,actual,className)` fatally asserts that `actual` is a MATLAB value whose class is the class specified by `className`.

`fatalAssertClass(fatalAssertable,actual,metaClass)` fatally asserts that `actual` is a MATLAB value whose class is the class specified by the `meta.class` instance `metaClass`.

`fatalAssertClass( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- The method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsOfClass;
  fatalAssertable.fatalAssertThat(actual, IsOfClass(className));
  fatalAssertable.fatalAssertThat(actual, IsOfClass(metaClass));
  ```

  There exists more functionality when using the `IsOfClass` constraint directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test.

className

Name of class, specified as a string.

metaClass

An instance of meta.class.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyClass`.

See Also

`fatalAssertThat` | `fatalAssertInstanceOf` |

matlab.unittest.TestCase.fatalAssertEmpty

Purpose

Fatally assert value is empty

Syntax

```
fatalAssertEmpty(fatalAssertable,actual)
fatalAssertEmpty(fatalAssertable,actual,diagnostic)
```

Description

`fatalAssertEmpty(fatalAssertable,actual)` fatally asserts that `actual` is an empty MATLAB value.

`fatalAssertEmpty(fatalAssertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  fatalAssertable.fatalAssertThat(actual, IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyEmpty`.

See Also

`fatalAssertThat` | `fatalAssertNotEmpty` | `isempty`

matlab.unittest.TestCase.fatalAssertEqual

Purpose

Fatally assert value is equal to specified value

Syntax

```
fatalAssertEqual(fatalAssertable,actual,expected)
fatalAssertEqual( ___ ,Name,Value)
fatalAssertEqual( ___ ,diagnostic)
```

Description

`fatalAssertEqual(fatalAssertable,actual,expected)` fatally asserts that `actual` is strictly equal to `expected` .

`fatalAssertEqual( ___ ,Name,Value)` fatally asserts equality with additional options specified by one or more `Name,Value` pair arguments.

`fatalAssertEqual( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to any of the following:

```
import matlab.unittest.constraints.IsEqualTo;
fatalAssertable.fatalAssertThat(actual, IsEqualTo(expected));

import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.AbsoluteTolerance;
fatalAssertable.fatalAssertThat(actual, IsEqualTo(expected, ...
    'Within', AbsoluteTolerance(abstol)));

import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.RelativeTolerance;
fatalAassertable.fatalAssertThat(actual, IsEqualTo(expected, ...
    'Within', RelativeTolerance(reltol)));

import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.AbsoluteTolerance;
import matlab.unittest.constraints.RelativeTolerance;
fatalAssertable.fatalAssertThat(actual, IsEqualTo(expected, ...
    'Within', AbsoluteTolerance(abstol) | RelativeTolerance(reltol)));
```

There exists more functionality when using the `IsEqualTo`, `RelativeTolerance`, and `IsEqualTo` constraints directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test.

expected

Expected value.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Name-Value Pair Arguments

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

'AbsTol'

'RelTol'

Examples

See examples for `verifyEqual`.

See Also

`fatalAssertThat` | `fatalAssertNotEqual` |

Purpose

Fatally assert function throws specified exception

Syntax

```
fatalAssertError(fatalAssertable,actual,identifier)
fatalAssertError(fatalAssertable,actual,metaClass)
fatalAssertError( ___ ,diagnostic)
```

Description

`fatalAssertError(fatalAssertable,actual,identifier)` fatally asserts that `actual` is a function handle that throws an exception with an error identifier that is equal to `identifier`.

`fatalAssertError(fatalAssertable,actual,metaClass)` fatally asserts that `actual` is a function handle that throws an exception whose type is defined by the meta.class instance specified in `metaClass`. This method does not require the instance to be an exact class match, but rather it must be in the specified class hierarchy, and that hierarchy must include the MException class..

`fatalAssertError( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Throws;
  fatalAssertable.fatalAssertThat(actual, Throws(identifier));
  fatalAssertable.fatalAssertThat(actual, Throws(metaClass));
  ```

  There exists more functionality when using the `Throws` constraint directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

identifier

Error identifier, specified as a string.

metaClass

An instance of meta.class.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyError`.

See Also

`fatalAssertThat` | `fatalAssertWarning` | `MException` | `error`

matlab.unittest.TestCase.fatalAssertFail

Purpose

Produce unconditional fatal assertion failure

Syntax

```
fatalAssertFail(fatalAssertable)
fatalAssertFail(fatalAssertable,diagnostic)
```

Description

`fatalAssertFail(fatalAssertable)` produces an unconditional fatal assertion failure when encountered.

`fatalAssertFail(fatalAssertable,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `matlab.unittest.TestCase.verifyFail`.

matlab.unittest.TestCase.fatalAssertFalse

Purpose

Fatally assert value is false

Syntax

```
fatalAssertFalse(fatalAssertable,actual)
fatalAssertFalse(fatalAssertable,actual,diagnostic)
```

Description

`fatalAssertFalse(fatalAssertable,actual)` fatally asserts that `actual` is a scalar logical with the value of false.

`fatalAssertFalse(fatalAssertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of false. Therefore, entities such as empty arrays, false valued arrays, and zero doubles produce failures when used in this method, despite these entities exhibiting "false-like" behavior such as bypassing the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsFalse;
  fatalAssertable.fatalAssertThat(actual, IsFalse());
  ```

  There exists more functionality when using the `IsFalse` constraint directly via fatalAssertThat.
- Unlike fatalAssertTrue, this method may create a new constraint for each call. For performance critical uses, consider using fatalAssertTrue.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyFalse`.

See Also

`fatalAssertThat` | `fatalAssertTrue` |

matlab.unittest.TestCase.fatalAssertGreaterThan

Purpose

Fatally assert value is greater than specified value

Syntax

```
fatalAssertGreaterThan(fatalAssertable,actual,floor)
fatalAssertGreaterThan(fatalAssertable,actual,floor,
    diagnostic)
```

Description

`fatalAssertGreaterThan(fatalAssertable,actual,floor)` fatally asserts that all elements of `actual` are greater than all the elements of `floor`.

`fatalAssertGreaterThan(fatalAssertable,actual,floor,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  matlab.unittest.constraints.IsGreaterThan;
  fatalAssertable.fatalAssertThat(actual, IsGreaterThan(floor));
  ```

  There exists more functionality when using the `IsGreaterThan` constraint directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test. `actual` must be the same size as `floor` unless either one is scalar, at which point scalar expansion occurs.

floor

Minimum value, exclusive.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyGreaterThan`.

See Also

`fatalAssertThat` | `fatalAssertGreaterThanOrEqual` | `fatalAssertLessThan` | `fatalAssertLessThanOrEqual` | `gt`

matlab.unittest.TestCase.fatalAssertGreaterThanOrEqual

Purpose

Fatally assert value is greater than or equal to specified value

Syntax

```
fatalAssertGreaterThanOrEqual(fatalAssertable,actual,floor)
fatalAssertGreaterThanOrEqual(fatalAssertable,actual,floor,
   diagnostic)
```

Description

`fatalAssertGreaterThanOrEqual(fatalAssertable,actual,floor)` fatally asserts that all elements of `actual` are greater than or equal to all the elements of `floor`.

`fatalAssertGreaterThanOrEqual(fatalAssertable,actual,floor,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsGreaterThanOrEqualTo;
  fatalAssertable.fatalAssertThat(actual, IsGreaterThanOrEqualTo(floor));
  ```

  There exists more functionality when using the `IsGreaterThanOrEqualTo` constraint directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test. `actual` must be the same size as `floor` unless either one is scalar, at which point scalar expansion occurs.

floor

Minimum value.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyGreaterThanOrEqual`.

See Also

`fatalAssertThat` | `fatalAssertGreaterThan` | `fatalAssertLessThanOrEqual` | `fatalAssertLessThan` | `ge`

matlab.unittest.TestCase.fatalAssertInstanceOf

Purpose

Fatally assert value is object of specified type

Syntax

```
fatalAssertInstanceOf(fatalAssertable,actual,className)
fatalAssertInstanceOf(fatalAssertable,actual,metaClass)
fatalAssertInstanceOf( ___ ,diagnostic)
```

Description

`fatalAssertInstanceOf(fatalAssertable,actual,className)` fatally asserts that `actual` is a MATLAB value whose class is the class specified by `className`.

`fatalAssertInstanceOf(fatalAssertable,actual,metaClass)` `actual` is a MATLAB value whose class is the class specified by the `meta.class` instance `metaClass`.

`fatalAssertInstanceOf( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

```
import matlab.unittest.constraints.IsInstanceOf;
fatalAssertable.fatalAssertThat(actual, IsInstanceOf(className));
fatalAssertable.fatalAssertThat(actual, IsInstanceOf(metaClass));
```

  There exists more functionality when using the `IsInstanceOf` constraint directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test.

className

Name of class, specified as a string.

metaClass

An instance of meta.class.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyInstanceOf`.

See Also

`fatalAssertThat` | `fatalAssertClass` | `isa`

matlab.unittest.TestCase.fatalAssertLength

Purpose

Fatally assert value has specified length

Syntax

```
fatalAssertLength(fatalAssertable,actual,expectedLength)
fatalAssertLength(fatalAssertable,actual,expectedLength,
   diagnostic)
```

Description

`fatalAssertLength(fatalAssertable,actual,expectedLength)` fatally asserts that `actual` is a MATLAB array whose length is `expectedLength`.

`fatalAssertLength(fatalAssertable,actual,expectedLength,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

```
import matlab.unittest.constraints.HasLength;
fatalAssertable.fatalAssertThat(actual, HasLength(expectedLength));
```

  There exists more functionality when using the `HasLength` constraint directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test.

expectedLength

The length of an array is defined as the largest dimension of that array.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyLength`.

See Also

`fatalAssertThat` | `fatalAssertSize` | `fatalAssertNumElements` | `length`

matlab.unittest.TestCase.fatalAssertLessThan

Purpose

Fatally assert value is less than specified value

Syntax

```
fatalAssertLessThan(fatalAssertable,actual,ceiling)
fatalAssertLessThan(fatalAssertable,actual,ceiling,
   diagnostic)
```

Description

`fatalAssertLessThan(fatalAssertable,actual,ceiling)` fatally asserts that all elements of `actual` are less than all the elements of `ceiling`.

`fatalAssertLessThan(fatalAssertable,actual,ceiling,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThan;
  assertable.assertThat(actual, IsLessThan(ceiling));
  ```

  There exists more functionality when using the `IsLessThan` constraint directly via assertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test. `actual` must be the same size as `ceiling` unless either one is scalar, at which point scalar expansion occurs.

ceiling

Maximum value, exclusive.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyLessThan`.

See Also

`assertThat` | `assertLessThanOrEqual` | `assertGreaterThan` | `assertGreaterThanOrEqual` | `lt`

matlab.unittest.TestCase.fatalAssertLessThanOrEqual

Purpose

Fatally assert value is less than or equal to specified value

Syntax

```
fatalAssertLessThanOrEqual(fatalAssertable,actual,ceiling)
fatalAssertLessThanOrEqual(fatalAssertable,actual,ceiling,
   diagnostic)
```

Description

`fatalAssertLessThanOrEqual(fatalAssertable,actual,ceiling)` fatally asserts that all elements of `actual` are less than or equal to all the elements of `ceiling`.

`fatalAssertLessThanOrEqual(fatalAssertable,actual,ceiling,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThanOrEqualTo;
  fatalAssertable.fatalAssertThat(actual, IsLessThanOrEqualTo(ceiling));
  ```

  There exists more functionality when using the `IsLessThanOrEqualTo` constraint directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test. `actual` must be the same size as `ceiling` unless either one is scalar, at which point scalar expansion occurs.

actual

The value to test.

ceiling

Maximum value.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyLessThanOrEqual`.

See Also

`fatalAssertThat` | `fatalAssertLessThan` | `fatalAssertGreaterThan` | `fatalAssertGreaterThanOrEqual` | `le`

matlab.unittest.TestCase.fatalAssertMatches

Purpose

Fatally assert string matches specified regular expression

Syntax

```
fatalAssertMatches(fatalAssertable,actual,expression)
fatalAssertMatches(fatalAssertable,actual,expression,
   diagnostic)
```

Description

`fatalAssertMatches(fatalAssertable,actual,expression)` fatally asserts that `actual` is a string that matches the regular expression defined by `expression`.

`fatalAssertMatches(fatalAssertable,actual,expression,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Matches;
  fatalAssertable.fatalAssertThat(actual, Matches(expression));
  ```

  There exists more functionality when using the `Matches` constraint directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test.

expression

The value to match, specified as a regular expression.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyMatches`.

See Also

`fatalAssertThat` | `fatalAssertSubstring` | `regexp`

matlab.unittest.TestCase.fatalAssertNotEmpty

Purpose

Fatally assert value is not empty

Syntax

```
fatalAssertNotEmpty(fatalAssertable,actual)
fatalAssertNotEmpty(fatalAssertable,actual,diagnostic)
```

Description

`fatalAssertNotEmpty(fatalAssertable,actual)` fatally asserts that `actual` is a non-empty MATLAB value.

`fatalAssertNotEmpty(fatalAssertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  fatalAssertable.fatalAssertThat(actual, ~IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyNotEmpty`.

See Also

`fatalAssertThat` | `fatalAssertEmpty` | `isempty`

matlab.unittest.TestCase.fatalAssertNotEqual

Purpose

Fatally assert value is not equal to specified value

Syntax

```
fatalAssertNotEqual(fatalAssertable,actual,notExpected)
fatalAssertNotEqual(fatalAssertable,actual,notExpected,
   diagnostic)
```

Description

`fatalAssertNotEqual(fatalAssertable,actual,notExpected)` fatally asserts that `actual` is not equal to `notExpected`.

`fatalAssertNotEqual(fatalAssertable,actual,notExpected,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEqualTo;
  fatalAssertable.fatalAssertThat(actual, ~IsEqualTo(notExpected));
  ```

  There exists more functionality when using the `IsEqualTo` constraint directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test.

notExpected

Value to compare.

Examples

See examples for `verifyNotEqual`.

See Also

`fatalAssertThat` | `fatalAssertEqual` |

matlab.unittest.TestCase.fatalAssertNotSameHandle

Purpose

Fatally assert value is not handle to specified instance

Syntax

```
fatalAssertNotSameHandle(fatalAssertable,actual,
   notExpectedHandle)
fatalAssertNotSameHandle(fatalAssertable,actual,
   notExpectedHandle,diagnostic)
```

Description

`fatalAssertNotSameHandle(fatalAssertable,actual,notExpectedHandle)` fatally asserts that `actual` is a different size and/or does not contain the same instances as the `notExpectedHandle` handle array.

`fatalAssertNotSameHandle(fatalAssertable,actual,notExpectedHandle,diag` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  fatalAssertable.fatalAssertThat(actual, ~IsSameHandleAs(notExpected
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test.

notExpectedHandle

The handle array to compare.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyNotSameHandle`.

See Also

`fatalAssertThat` | `fatalAssertSameHandle` |

Purpose

Fatally assert value has specified element count

Syntax

```
fatalAssertNumElements(fatalAssertable,actual,
   expectedElementCount)
fatalAssertNumElements(fatalAssertable,actual,
   expectedElementCount,diagnostic)
```

Description

`fatalAssertNumElements(fatalAssertable,actual,expectedElementCount)` fatally asserts that `actual` is a MATLAB array with `expectedElementCount` number of elements.

`fatalAssertNumElements(fatalAssertable,actual,expectedElementCount,dia` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

```
import matlab.unittest.constraints.HasElementCount;
fatalAssertable.fatalAssertThat(actual, HasElementCount(expectedEler
```

  There exists more functionality when using the `HasElementCount` constraint directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test.

expectedElementCount

The expected number of elements in the array.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyNumElements`.

See Also

`fatalAssertThat` | `fatalAssertSize` | `fatalAssertLength` | `numel`

Purpose

Fatally assert function returns true when evaluated

Syntax

```
fatalAssertReturnsTrue(fatalAssertable,actual)
fatalAssertReturnsTrue(fatalAssertable,actual,diagnostic)
```

Description

`fatalAssertReturnsTrue(fatalAssertable,actual)` fatally asserts that `actual` is a function handle that returns a scalar logical whose value is true.

`fatalAssertReturnsTrue(fatalAssertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- It is a shortcut for quick custom comparison functionality that can be defined quickly, and possibly inline. It can be preferable over simply evaluating the function directly and using fatalAssertTrue because the function handle will be shown in the diagnostics, thus providing more insight into the failure condition which is lost when using fatalAssertTrue.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ReturnsTrue;
  fatalAssertable.fatalAssertThat(actual, ReturnsTrue());
  ```

  There exists more functionality when using the `ReturnsTrue` constraint directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The function handle to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyReturnsTrue`.

See Also

`fatalAssertThat` | `fatalAssertTrue` |

Purpose

Fatally assert two values are handles to same instance

Syntax

```
fatalAssertSameHandle(fatalAssertable,actual,expectedHandle)
fatalAssertSameHandle(fatalAssertable,actual,expectedHandle,
   diagnostic)
```

Description

`fatalAssertSameHandle(fatalAssertable,actual,expectedHandle)` fatally asserts that `actual` is the same size and contains the same instances as the `expectedHandle` handle array.

`fatalAssertSameHandle(fatalAssertable,actual,expectedHandle,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  fatalAssertable.fatalAssertThat(actual, IsSameHandleAs(expectedHandl
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test.

expectedHandle

The expected handle array.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifySameHandle`.

See Also

`fatalAssertThat` | `fatalAssertNotSameHandle` | `handle`

Purpose

Fatally assert value has specified size

Syntax

```
fatalAssertSize(fatalAssertable,actual,expectedSize)
fatalAssertSize(fatalAssertable,actual,expectedSize,
   diagnostic)
```

Description

`fatalAssertSize(fatalAssertable,actual,expectedSize)` fatally asserts that `actual` is a MATLAB array whose size is `expectedSize`.

`fatalAssertSize(fatalAssertable,actual,expectedSize,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasSize;
  fatalAssertable.fatalAssertThat(actual, HasSize(expectedSize));
  ```

  There exists more functionality when using the `HasSize` constraint directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test.

expectedSize

The expected sizes of each dimension the array.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifySize`.

See Also

`fatalAssertThat` | `fatalAssertLength` | `fatalAssertNumElements` | `size`

Purpose

Fatally assert string contains specified string

Syntax

```
fatalAssertSubstring(fatalAssertable,actual,substring)
fatalAssertSubstring(fatalAssertable,actual,substring,
   diagnostic)
```

Description

`fatalAssertSubstring(fatalAssertable,actual,substring)` fatally asserts that `actual` is a string that contains `substring`.

`fatalAssertSubstring(fatalAssertable,actual,substring,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

```
import matlab.unittest.constraints.ContainsSubstring;
fatalAssertable.fatalAssertThat(actual,...
ContainsSubstring(substring));
```

  There exists more functionality when using the `ContainsSubstring` constraint directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test.

substring

The value to match, specified as a string.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifySubstring`.

See Also

`fatalAssertThat` | `fatalAssertMatches` | `strfind`

Purpose

Fatally assert value meets specified constraint

Syntax

```
fatalAssertThat(fatalAssertable,actual,constraint)
fatalAssertThat(fatalAssertable,actual,constraint,diagnostic)
```

Description

`fatalAssertThat(fatalAssertable,actual,constraint)` fatally asserts that `actual` is a value that satisfies the `constraint` provided.

If the constraint is not satisfied, a fatal assertion failure is produced utilizing only the framework diagnostic generated by the `constraint`.

`fatalAssertThat(fatalAssertable,actual,constraint,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

When using this signature, both the diagnostic information contained within `diagnostic` is used in addition to the diagnostic information provided by the `constraint`.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test.

constraint

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyThat`.

Purpose

Fatally assert value is true

Syntax

```
fatalAssertTrue(fatalAssertable,actual)
fatalAssertTrue(fatalAssertable,actual,diagnostic)
```

Description

`fatalAssertTrue(fatalAssertable,actual)` fatally asserts that `actual` is a scalar logical with the value of true.

`fatalAssertTrue(fatalAssertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of true. Therefore, entities such as true valued arrays and nonzero doubles produce qualification failures when used in this method, despite these entities exhibiting "true-like" behavior such as triggering the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsTrue;
  fatalAssertable.fatalAssertThat(actual, IsTrue());
  ```

  There exists more functionality when using the `IsTrue` constraint directly via fatalAssertThat.

  However, this method is optimized for performance and does not construct a new `IsTrue` constraint for each call. Sometimes such use can come at the expense of less diagnostic information. Use the fatalAssertReturnsTrue method for a similar approach which may provide better diagnostic information.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The value to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

See examples for `verifyTrue`.

See Also

`fatalAssertThat` | `fatalAssertFalse` | `fatalAssertReturnsTrue` |

Purpose

Fatally assert function issues specified warning

Syntax

```
fatalAssertWarning(fatalAssertable,actual,warningID)
fatalAssertWarning(fatalAssertable,actual,warningID,
   diagnostic)
[output1,...,outputN] = fatalAssertWarning( ___ )
```

Description

`fatalAssertWarning(fatalAssertable,actual,warningID)` fatally asserts that `actual` issues a warning with the identifier `warningID`.

`fatalAssertWarning(fatalAssertable,actual,warningID,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`[output1,...,outputN] = fatalAssertWarning( ___ )` also returns the output arguments `output1,...,outputN` that are produced when invoking `actual`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesWarnings;
  fatalAssertable.fatalAssertThat(actual, IssuesWarnings({warningID}))
  ```

  There exists more functionality when using the `IssuesWarnings` constraint directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The function handle to test.

warningID

Warning ID, specified as a string.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Output Arguments

output1,...,outputN

Output arguments, 1 through n (if any), from `actual`, returned as any type. The argument type is specified by the `actual` argument list.

Examples

See examples for `verifyWarning`.

See Also

`fatalAssertThat` | `fatalAssertError` | `fatalAssertWarningFree` | `warning`

Purpose

Fatally assert function issues no warnings

Syntax

```
fatalAssertWarningFree(fatalAssertable,actual)
fatalAssertWarningFree(fatalAssertable,actual,diagnostic)
[output1,...,outputN] = fatalAssertWarningFree( ___ )
```

Description

`fatalAssertWarningFree(fatalAssertable,actual)` fatally asserts that `actual` is a function handle that issues no warnings.

`fatalAssertWarningFree(fatalAssertable,actual,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`[output1,...,outputN] = fatalAssertWarningFree( ___ )` also returns the output arguments `output1,...,outputN` that are produced when invoking `actual`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesNoWarnings;
  fatalAssertable.fatalAssertThat(actual, IssuesNoWarnings());
  ```

  There exists more functionality when using the `IssuesNoWarnings` constraint directly via fatalAssertThat.

Input Arguments

fatalAssertable

The matlab.unittest.TestCase instance which is used to pass or fail the fatal assertion in conjunction with the test running framework.

actual

The function handle to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Output Arguments

output1,...,outputN

Output arguments, 1 through n (if any), from `actual`, returned as any type. The argument type is specified by the `actual` argument list.

Examples

See examples for `verifyWarningFree`.

See Also

`fatalAssertThat` | `fatalAssertWarning` | `warning`

Purpose

Run `TestCase` test

Syntax

```
result = run(testCase)
result = run(testCase,testMethod)
```

Description

`result = run(testCase)` uses `testCase` as a prototype to run a TestSuite array created from all test methods in the class defining `testCase`. This suite is run using a TestRunner object configured for text output.

`result = run(testCase,testMethod)` uses `testCase` as a prototype to run a TestSuite array created from `testMethod`. This test is run using a TestRunner object configured for text output.

This is a convenience method to allow interactive experimentation of TestCase classes in MATLAB, yet running the tests contained in them using a supported TestRunner object.

Input Arguments

testCase

matlab.unittest.TestCase instance

testMethod

Name of desired test method, specified as one of the following:

- string
- `meta.method` instance

The method must correspond to a valid `Test` method of the `testCase` instance.

Output Arguments

result

A matlab.unittest.TestResult object containing the result of the test run.

Examples

Run Test Directly from Test Case

Add the `FigurePropertiesTest.m` test case file to a folder on your MATLAB path.

```
classdef FigurePropertiesTest < matlab.unittest.TestCase

    properties
        TestFigure
    end

    methods(TestMethodSetup)
        function createFigure(testCase)
            %comment
            testCase.TestFigure = figure;
        end
    end

    methods(TestMethodTeardown)
        function closeFigure(testCase)
            close(testCase.TestFigure);
        end
    end

    methods(Test)

        function defaultCurrentPoint(testCase)

            cp = get(testCase.TestFigure, 'CurrentPoint');
            testCase.verifyEqual(cp, [0 0], ...
                'Default current point is incorrect')
        end

        function defaultCurrentObject(testCase)
            import matlab.unittest.constraints.IsEmpty;

            co = get(testCase.TestFigure, 'CurrentObject');
```

```
            testCase.verifyThat(co, IsEmpty, ...
                'Default current object should be empty');
        end

    end

end
```

Create a testcase object.

```
tc = FigurePropertiesTest;
```

Run the tests.

```
tc.run;
```

```
Running FigurePropertiesTest
..
Done FigurePropertiesTest
__________
```

All tests passed.

See Also

matlab.unittest.TestSuite.run | matlab.unittest.TestRunner.run |

matlab.unittest.TestCase.verifyClass

Purpose

Verify exact class of specified value

Syntax

```
verifyClass(verifiable,actual,className)
verifyClass(verifiable,actual,metaClass)
verifyClass(___,diagnostic)
```

Description

`verifyClass(verifiable,actual,className)` verifies that `actual` is a MATLAB value whose class is the class specified by `className`.

`verifyClass(verifiable,actual,metaClass)` verifies that `actual` is a MATLAB value whose class is the class specified by the `meta.class` instance `metaClass`. The instance must be an exact class match. Use verifyInstanceOf to verify inclusion in a class hierarchy.

`verifyClass(___,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- The method is functionally equivalent to the following methods:

  ```
  import matlab.unittest.constraints.IsOfClass;
  verifiable.verifyThat(actual, IsOfClass(className));
  verifiable.verifyThat(actual, IsOfClass(metaClass));
  ```

  There exists more functionality when using the `IsOfClass` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The value to test.

className

Name of class, specified as a string.

metaClass

An instance of meta.class.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Test a Class

These interactive tests verify the class of the number, 5.

Create a TestCase object and the value to test.

```
testCase = matlab.unittest.TestCase;
actvalue = 5;
```

Verify class of `actvalue` is `double`.

```
testCase.verifyClass(actvalue, 'double');
```

```
Interactive verification passed.
```

Verify class of `actvalue` is `char`.

```
testCase.verifyClass(actvalue, 'char');
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
```

```
verifyClass failed.
--> The value's class is incorrect.

    Actual Class:
        double
    Expected Class:
        char

Actual Value:
         5
```

Test fails.

Test a Function Handle

These interactive tests verify function handles, specified as a meta.class instance, `?function_handle`.

Create a TestCase object.

```
testCase = matlab.unittest.TestCase;
```

Create a function handle.

```
fh = @sin;
testCase.verifyClass(fh, ?function_handle);
```

```
Interactive verification passed.
```

Test the function name.

```
fh = 'sin';
testCase.verifyClass(fh, ?function_handle);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
```

```
verifyClass failed.
--> The value's class is incorrect.

    Actual Class:
        char
    Expected Class:
        function_handle

Actual Value:
    sin
```

Test fails.

Test a Derived Class

Verify that a derived class is not the same class as its base class.

Create a class, `BaseExample`.

```
classdef BaseExample
end
```

Create a derived class, `DerivedExample`.

```
classdef DerivedExample < BaseExample
end
```

Verify the classes are not equal.

```
testCase.verifyClass(DerivedExample(), ?BaseExample);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyClass failed.
--> The value's class is incorrect.
```

```
    Actual Class:
        DerivedExample
    Expected Class:
        BaseExample

Actual Value:
      DerivedExample with no properties.
```

Test fails.

Test Class of Output Value

Use `verifyClass` to test the `add5` function returns a `double` value.

Function for unit testing:

```
function res = add5(x)
%ADD5 Increment input by 5.
if ~isa(x,'numeric')
    error('add5:InputMustBeNumeric','Input must be numeric.');
end
res = x + 5;
end
```

TestCase class containing test methods:

```
classdef Add5Test < matlab.unittest.TestCase
    methods (Test)
        function testDoubleOut(testCase)
            actOutput = add5(1);
            testCase.verifyClass(actOutput,'double');
        end
        function testNonNumericInput(testCase)
            testCase.verifyError(@()add5('0'),'add5:InputMustBeNumeric');
        end
    end
end
```

Create a test suite from the `Add5Test` class file.

```
suite = matlab.unittest.TestSuite.fromFile('Add5Test.m');

result = run(suite);

Running Add5Test
..
Done Add5Test
__________
```

See Also

verifyThat | verifyInstanceOf |

matlab.unittest.TestCase.verifyEmpty

Purpose

Verify value is empty

Syntax

```
verifyEmpty(verifiable,actual)
verifyEmpty(___,diagnostic)
```

Description

`verifyEmpty(verifiable,actual)` verifies that `actual` is an empty MATLAB value.

`verifyEmpty(___,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsEmpty;
  verifiable.verifyThat(actual, IsEmpty());
  ```

  There exists more functionality when using the `IsEmpty` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The value to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Test for Empty Strings

Create a TestCase object.

```
testCase = matlab.unittest.TestCase;

testCase.verifyEmpty('');

Interactive verification passed.
```

Test for Empty Arrays

An array with any zero dimension is empty.

```
testCase.verifyEmpty(ones(2, 5, 0, 3));

Interactive verification passed.

testCase.verifyEmpty([2 3], 'Array is not empty.');

Interactive verification failed.

----------------
Test Diagnostic:
----------------
Array is not empty.

---------------------
Framework Diagnostic:
---------------------
verifyEmpty failed.
--> The value must be empty.
--> The value has a size of [1  2].

Actual Value:
         2     3
```

Test failed.

Test for Empty Cell Arrays

Test empty cell array, `{}`.

```
testCase.verifyEmpty({}, 'Cell array is not empty.');
```

```
Interactive verification passed.
```

A cell array of empty arrays is not empty.

```
testCase.verifyEmpty({[], [], []}, 'Cell array is not empty.');
```

```
Interactive verification failed.

----------------
Test Diagnostic:
----------------
Cell array is not empty.

---------------------
Framework Diagnostic:
---------------------
verifyEmpty failed.
--> The value must be empty.
--> The value has a size of [1  3].

Actual Value:
        []    []    []
```

Test failed.

Test for Empty Test Suite

Test for empty object, `emptyTestSuite`.

```
emptyTestSuite = matlab.unittest.TestSuite.emtpy;
testCase.verifyEmpty(emptyTestSuite);
```

```
Interactive verification passed.
```

See Also

verifyThat | matlab.unittest.TestCase.verifyNotEmpty | isempty

matlab.unittest.TestCase.verifyEqual

Purpose

Verify value is equal to specified value

Syntax

```
verifyEqual(verifiable,actual,expected)
verifyEqual(___,Name,Value)
verifyEqual(___,diagnostic)
```

Description

`verifyEqual(verifiable,actual,expected)`) verifies that `actual` is strictly equal to `expected`.

`verifyEqual(___,Name,Value)` verifies equality with additional options specified by one or more `Name,Value` pair arguments.

`verifyEqual(___,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure

Tips

- This method is functionally equivalent to any of the following:

```
import matlab.unittest.constraints.IsEqualTo;
verifiable.verifyThat(actual, IsEqualTo(expected));

import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.AbsoluteTolerance;
verifiable.verifyThat(actual, IsEqualTo(expected, ...
    'Within', AbsoluteTolerance(abstol)));

import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.RelativeTolerance;
verifiable.verifyThat(actual, IsEqualTo(expected, ...
    'Within', RelativeTolerance(reltol)));

import matlab.unittest.constraints.IsEqualTo;
import matlab.unittest.constraints.AbsoluteTolerance;
import matlab.unittest.constraints.RelativeTolerance;
verifiable.verifyThat(actual, IsEqualTo(expected, ...
    'Within', AbsoluteTolerance(abstol) | RelativeTolerance(reltol)));
```

There exists more functionality when using the `IsEqualTo`, `RelativeTolerance`, and `IsEqualTo` constraints directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The value to test.

expected

Expected value.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Name-Value Pair Arguments

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

'AbsTol'

'RelTol'

Examples

Comparing Numeric Values

Numeric values are equivalent if they are of the same class with equivalent size, complexity, and sparsity.

Create a TestCase object for interactive testing.

```
testCase = matlab.unittest.TestCase;
```

A value is equal to itself.

```
testCase.verifyEqual(5, 5);
```

```
Interactive verification passed.
```

Values must have equal sizes.

```
testCase.verifyEqual([5 5], 5);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyEqual failed.
--> NumericComparator failed.
    --> Sizes do not match.

    Actual double size:
             1     2
    Expected double size:
             1     1

Actual Value:
         5     5
Expected Value:
         5
```

Test failed.

Test Classes

```
testCase.verifyEqual(int8(5), int16(5));
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyEqual failed.
--> NumericComparator failed.
    --> Classes do not match.

    Actual Class:
        int8
    Expected Class:
        int16

Actual Value:
        5
Expected Value:
          5
```

Test failed.

Test Cell Arrays

Each element of a cell array must be equal in value, class, and size.

```
testCase.verifyEqual({'cell', struct, 5}, {'cell', struct, 5});
```

```
Interactive verification passed.
```

Test Numeric Tolerances

```
testCase.verifyEqual(4.95, 5);
```

```
Interactive verification failed.
```

```
----------------
Test Diagnostic:
----------------
4.95 is not equal to 5

---------------------
Framework Diagnostic:
---------------------
verifyEqual failed.
--> NumericComparator failed.
    --> The values are not equal using "isequaln".

Actual Value:
       4.950000000000000
Expected Value:
          5
```

Test failed.

```
testCase.verifyEqual(1.5, 2, 'AbsTol', 1)
```

```
Interactive verification passed.
```

```
testCase.verifyEqual(1.5, 2, 'RelTol', 0.1, ...
    'Difference between actual and expected exceeds relative tolerance')
```

```
Interactive verification failed.

----------------
Test Diagnostic:
----------------
Difference between actual and expected exceeds relative tolerance

---------------------
Framework Diagnostic:
---------------------
verifyEqual failed.
```

```
--> NumericComparator failed.
    --> The values are not equal using "isequaln".
    --> RelativeTolerance failed.
        --> The value was not within relative tolerance.

            Tolerance Definition:
                abs(expected - actual) <= tolerance .* abs(expected)
            Tolerance Value:
                   0.100000000000000

Actual Value:
       1.500000000000000
Expected Value:
         2
```

Test failed.

See Also

verifyThat | verifyNotEqual |

matlab.unittest.TestCase.verifyError

Purpose

Verify function throws specified exception

Syntax

```
verifyError(verifiable,actual,identifier)
verifyError(verifiable,actual,metaClass)
verifyError( ___ ,diagnostic)
```

Description

`verifyError(verifiable,actual,identifier)` verifies that `actual` is a function handle that throws an exception with an error identifier that is equal to `identifier`.

`verifyError(verifiable,actual,metaClass)` verifies that `actual` is a function handle that throws an exception whose type is defined by the meta.class instance specified in `metaClass`. This method does not require the instance to be an exact class match, but rather it must be in the specified class hierarchy, and that hierarchy must include the MException class.

`verifyError( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Throws;
  verifiable.verifyThat(actual, Throws(identifier));
  verifiable.verifyThat(actual, Throws(metaClass));
  ```

  There exists more functionality when using the `Throws` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The value to test.

identifier

Error identifier, specified as a string.

metaClass

An instance of meta.class.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Test for Error IDs

```
% Passing scenarios
%%%%%%%%%%%%%%%%%%%%
testCase.verifyError(@() error('SOME:error:id','Error!'), 'SOME:error:
testCase.verifyError(@testCase.assertFail, ...
    ?matlab.unittest.qualifications.AssertionFailedException);

% Failing scenarios
%%%%%%%%%%%%%%%%%%%%
testCase.verifyError(5, 'some:id', '5 is not a function handle');
testCase.verifyError(@testCase.verifyFail, ...
    ?matlab.unittest.qualifications.AssertionFailedException, ...
    'Verifications dont throw exceptions.');
testCase.verifyError(@() error('SOME:id'), 'OTHER:id', 'Wrong id');
testCase.verifyError(@() error('whoops'), ...
    ?matlab.unittest.qualifications.AssertionFailedException, ...
    'Wrong type of exception thrown');
```

Test Error Condition

Create `testNonNumericInput` to test if function throws expected error message, `add5:InputMustBeNumeric`, for unexpected condition, input is `char`.

Function for unit testing:

```
function res = add5(x)
%ADD5 Increment input by 5.
if ~isa(x,'numeric')
    error('add5:InputMustBeNumeric','Input must be numeric.');
end
res = x + 5;
end
```

TestCase class containing test methods:

```
classdef Add5Test < matlab.unittest.TestCase
    methods (Test)
        function testDoubleOut(testCase)
            actOutput = add5(1);
            testCase.verifyClass(actOutput,'double');
        end
        function testNonNumericInput(testCase)
            testCase.verifyError(@()add5('0'),'add5:InputMustBeNumeric');
        end
    end
end
```

Create a test suite from the `Add5Test` class file.

```
suite = matlab.unittest.TestSuite.fromFile('Add5Test.m');

result = run(suite);
```

```
Running Add5Test
..
Done Add5Test
```

See Also

`verifyThat` | `verifyWarning` | `MException` | `error`

matlab.unittest.TestCase.verifyFail

Purpose

Produce unconditional verification failure

Syntax

```
verifyFail(verifiable)
verifyFail(verifiable,diagnostic)
```

Description

`verifyFail(verifiable)` produces an unconditional verification failure when encountered.

`verifyFail(verifiable,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

An example of where this method may be used is in a callback function that should not be executed in a given scenario. A test can confirm this does not occur by unconditionally performing a failure if the code path is reached.

Assume a handle class MyHandle with a SomethingHappened event.

```
classdef MyHandle < handle
    events
```

```
        SomethingHappened
    end
end
```

The following is a test method inside a TestCase class.

```
methods(Test)
    function testDisabledListeners(testCase)

        h = MyHandle;

        % Add a listener to a test helper method
        listener = h.addlistener('SomethingHappened', ...
            @testCase.shouldNotGetCalled);

        % Passing scenario (code path is not reached)
        %%%%%%%%%%%%%%%%%%%%
        % Disabled listener should not invoke callbacks
        listener.Enabled = false;
        h.notify('SomethingHappened');

        % Failing scenario (code path is reached)
        %%%%%%%%%%%%%%%%%%%%
        % Enabled listener invoke callback and fail
        listener.Enabled = true;
        h.notify('SomethingHappened');
    end
end
```

matlab.unittest.TestCase.verifyFalse

Purpose

Verify value is false

Syntax

```
verifyFalse(verifiable,actual)
verifyFalse( ___ ,diagnostic)
```

Description

`verifyFalse(verifiable,actual)` verifies that `actual` is a scalar logical with the value of false.

`verifyFalse( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of false. Therefore, entities such as empty arrays, false valued arrays, and zero doubles produce failures when used in this method, despite these entities exhibiting "false-like" behavior such as bypassing the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsFalse;
  verifiable.verifyThat(actual, IsFalse());
  ```

  There exists more functionality when using the `IsFalse` constraint directly via verifyThat.
- Unlike verifyTrue, this method may create a new constraint for each call. For performance critical uses, consider using verifyTrue.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The value to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Test MATLAB Logical Functions

Create a TestCase object for interactive testing.

```
testCase = matlab.unittest.TestCase;
```

Test `true`.

```
testCase.verifyFalse(true);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyFalse failed.
--> The value must evaluate to "false".

Actual Value:
         1
```

Test failed.

Test `false`.

```
testCase.verifyFalse(false);
```

```
Interactive verification passed.
```

Test the Value 0

The number `0` is a `double` value, not a logical value.

```
testCase.verifyFalse(0);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyFalse failed.
--> The value must be logical. It is of type "double".

Actual Value:
         0
```

Test failed.

Test Array of Logical Values

To be false, the value must be scalar.

```
testCase.verifyFalse([false false false]);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyFalse failed.
--> The value must be scalar. It has a size of [1  3].

Actual Value:
         0     0     0
```

Test failed.

Test an array of mixed logical values.

```
testCase.verifyFalse([false true false], ...
    'A mixed array of logicals is not the one false value');
```

```
Interactive verification failed.

----------------
Test Diagnostic:
----------------
A mixed array of logicals is not the one false value

---------------------
Framework Diagnostic:
---------------------
verifyFalse failed.
--> The value must be scalar. It has a size of [1  3].

Actual Value:
         0     1     0
```

Test failed.

See Also

verifyThat | verifyTrue |

matlab.unittest.TestCase.verifyGreaterThan

Purpose

Verify value is greater than specified value

Syntax

```
verifyGreaterThan(verifiable,actual,floor)
verifyGreaterThan( ___ ,diagnostic)
```

Description

`verifyGreaterThan(verifiable,actual,floor)` verifies that all elements of `actual` are greater than all the elements of `floor`.

`verifyGreaterThan( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  matlab.unittest.constraints.IsGreaterThan;
  verifiable.verifyThat(actual, IsGreaterThan(floor));
  ```

  There exists more functionality when using the `IsGreaterThan` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The value to test. `actual` must be the same size as `floor` unless either one is scalar, at which point scalar expansion occurs.

floor

Minimum value, exclusive.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Compare Two Numbers

Create a TestCase object for interactive testing.

```
testCase = matlab.unittest.TestCase;
```

Verify 3 is greater than 2.

```
testCase.verifyGreaterThan(3, 2);
```

```
Interactive verification passed.
```

Test if 5 is greater than 9.

```
testCase.verifyGreaterThan(5, 9);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyGreaterThan failed.
--> The value must be greater than the minimum value.

Actual Value:
         5
Minimum Value (Exclusive):
         9
```

Test failed.

Compare an Array to a Scalar

Test if each element is greater than the `FLOOR` value, `2`.

```
testCase.verifyGreaterThan([5 6 7], 2);
```

```
Interactive verification passed.
```

Test if value `5` is greater than each element in the `FLOOR` array, `[1 2 3]`.

```
testCase.verifyGreaterThan(5, [1 2 3]);
```

```
Interactive verification passed.
```

Test if each element in the matrix is greater than the `FLOOR` value, `4`.

```
testCase.verifyGreaterThan([1 2 3; 4 5 6], 4);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyGreaterThan failed.
--> Each element must be greater than the minimum value.

    Failing Indices:
             1     2     3     5

Actual Value:
         1     2     3
         4     5     6
Minimum Value (Exclusive):
         4
```

Test failed.

Compare Arrays

Test if each element is greater than each corresponding element of the `FLOOR` array, `[4 -9 0]`.

```
testCase.verifyGreaterThan([5 -3 2], [4 -9 0]);
```

```
Interactive verification passed.
```

Compare an array to itself.

```
testCase.verifyGreaterThan(eye(2), eye(2));
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyGreaterThan failed.
--> Each element must be greater than each corresponding element of th

    Failing Indices:
             1     2     3     4

Actual Value:
         1     0
         0     1
Minimum Value (Exclusive):
         1     0
         0     1
```

Test failed.

See Also

`verifyThat` | `verifyGreaterThanOrEqual` | `verifyLessThanOrEqual` | `verifyLessThan` | `gt`

matlab.unittest.TestCase.verifyGreaterThanOrEqual

Purpose

Verify value is greater than or equal to specified value

Syntax

```
verifyGreaterThanOrEqual(verifiable,actual,floor)
verifyGreaterThanOrEqual(___,diagnostic)
```

Description

`verifyGreaterThanOrEqual(verifiable,actual,floor)` that all elements of `actual` are greater than or equal to all the elements of `floor`.

`verifyGreaterThanOrEqual(___,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsGreaterThanOrEqualTo;
  verifiable.verifyThat(actual, IsGreaterThanOrEqualTo(floor));
  ```

  There exists more functionality when using the `IsGreaterThanOrEqualTo` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The value to test. `actual` must be the same size as `floor` unless either one is scalar, at which point scalar expansion occurs.

floor

Minimum value.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Compare Two Numbers

Create a TestCase object for interactive testing.

```
testCase = matlab.unittest.TestCase;
```

Verify 3 is greater than 2.

```
testCase.verifyGreaterThanOrEqual(3, 2);
```

```
Interactive verification passed.
```

Verify 3 is greater than or equal to 3.

```
testCase.verifyGreaterThanOrEqual(3, 3);
```

```
Interactive verification passed.
```

Test if 5 is greater than 9.

```
testCase.verifyGreaterThanOrEqual(5, 9);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyGreaterThanOrEqual failed.
--> The value must be greater than or equal to the minimum value.

Actual Value:
         5
```

```
Minimum Value (Inclusive):
         9
```

Test failed.

Compare an Array to a Scalar

Test if each element is greater than or equal to the `FLOOR` value, `2`.

```
testCase.verifyGreaterThanOrEqual([5 2 7], 2);
```

```
Interactive verification passed.
```

Test if each element in the matrix is greater than or equal to the `FLOOR` value, `4`.

```
testCase.verifyGreaterThanOrEqual([1 2 3; 4 5 6], 4);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyGreaterThanOrEqual failed.
--> Each element must be greater than or equal to the minimum value.

    Failing Indices:
             1     3     5

Actual Value:
         1     2     3
         4     5     6
Minimum Value (Inclusive):
         4
```

Compare Arrays

Test if each element is greater than or equal to each corresponding element of the `FLOOR` array, `[4 -3 0]`.

```
testCase.verifyGreaterThanOrEqual([5 -3 2], [4 -3 0]);
```

```
Interactive verification passed.
```

Compare an array to itself.

```
testCase.verifyGreaterThanOrEqual(eye(2), eye(2));
```

```
Interactive verification passed.
```

See Also

`verifyThat` | `verifyGreaterThan` | `verifyLessThan` | `verifyLessThanOrEqual` | `ge`

matlab.unittest.TestCase.verifyInstanceOf

Purpose

Verify value is object of specified type

Syntax

```
verifyInstanceOf(verifiable,actual,className)
verifyInstanceOf(verifiable,actual,metaClass)
verifyInstanceOf( ___ ,diagnostic)
```

Description

`verifyInstanceOf(verifiable,actual,className)` verifies that `actual` is a MATLAB value whose class is the class specified by `className`.

`verifyInstanceOf(verifiable,actual,metaClass)` `actual` is a MATLAB value whose class is the class specified by the `meta.class` instance `metaClass`.

`verifyInstanceOf( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsInstanceOf;
  verifiable.verifyThat(actual, IsInstanceOf(className));
  verifiable.verifyThat(actual, IsInstanceOf(metaClass));
  ```

  There exists more functionality when using the `IsInstanceOf` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The value to test.

className

Name of class, specified as a string.

metaClass

An instance of meta.class.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Test a Class

These interactive tests verify the class of the number, 5.

Create a TestCase object and the value to test.

```
testCase = matlab.unittest.TestCase;
actvalue = 5;
```

Verify `actvalue` is an instance of class `double`.

```
testCase.verifyInstanceOf(actvalue, 'double');
```

```
Interactive verification passed.
```

Verify if `actvalue` is an instance of `char`.

```
testCase.verifyInstanceOf(5, 'char');
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyInstanceOf failed.
```

```
--> The value must be an instance of the expected type.

    Actual Class:
        double
    Expected Type:
        char

Actual Value:
         5
```

Test failed.

Test a Function Handle

These tests verify function handles, specified as a meta.class instance, `?function_handle`.

Create a function handle.

```
fh = @sin;
testCase.verifyInstanceOf(fh, ?function_handle);
```

```
Interactive verification passed.
```

Test the function name.

```
fh = 'sin';
testCase.verifyInstanceOf(fh, ?function_handle);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyInstanceOf failed.
--> The value must be an instance of the expected type.

    Actual Class:
        char
```

```
    Expected Type:
        function_handle

Actual Value:
    sin
```

Test failed.

Test a Derived Class

Verify that a derived class is not the same class as its base class.

Create a class, `BaseExample`.

```
classdef BaseExample
end
```

Create a derived class, `DerivedExample`.

```
classdef DerivedExample < BaseExample
end
```

Verify `DerivedExample` is an instance of `BaseExample`.

```
testCase.verifyInstanceOf(DerivedExample(), ?BaseExample);
```

```
Interactive verification passed.
```

Verify `BaseExample` is not an instance of `DerivedExample`.

```
testCase.verifyInstanceOf(BaseExample(), ?DerivedExample);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyInstanceOf failed.
--> The value must be an instance of the expected type.
```

```
    Actual Class:
        BaseExample
    Expected Type:
        DerivedExample

Actual Value:
      BaseExample with no properties.
```

Test failed.

See Also

`verifyThat` | `verifyClass` | `isa`

Purpose

Verify value has specified length

Syntax

```
verifyLength(verifiable,actual,expectedLength)
verifyLength( ___ ,diagnostic)
```

Description

`verifyLength(verifiable,actual,expectedLength)` verifies that `actual` is a MATLAB array whose length is `expectedLength`.

`verifyLength( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

```
import matlab.unittest.constraints.HasLength;
verifiable.verifyThat(actual, HasLength(expectedLength));
```

  There exists more functionality when using the `HasLength` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The value to test.

expectedLength

The length of an array is defined as the largest dimension of that array.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Test Array Lengths

Create a TestCase object for interactive testing.

```
testCase = matlab.unittest.TestCase;
```

Verify length of array is the expected value, 5.

```
testCase.verifyLength(ones(2, 5, 3), 5, 'User diagnostic');
```

```
Interactive verification passed.
```

Length of array is not the expected value, 3.

```
testCase.verifyLength([2 3], 3);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyLength failed.
--> The array has an incorrect length.

    Actual Length:
             2
    Expected Length:
             3

Actual Array:
         2     3
```

Test failed.

The length of a 2x3 array is `3`.

```
testCase.verifyLength([1 2 3; 4 5 6], 3);
```

```
Interactive verification passed.
```

Verify the length of a 2x3 array is not the number of elements, `6`.

```
testCase.verifyLength([1 2 3; 4 5 6], 6);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyLength failed.
--> The array has an incorrect length.

    Actual Length:
             3
    Expected Length:
             6

Actual Array:
         1     2     3
         4     5     6
```

Test failed.

```
testCase.verifyLength(eye(2), 4);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
```

```
verifyLength failed.
--> The array has an incorrect length.

    Actual Length:
             2
    Expected Length:
             4

Actual Array:
         1     0
         0     1
```

Test Cell Array Lengths

```
testCase.verifyLength({'somestring', 'someotherstring'}, 2);
```

```
Interactive verification passed.
```

See Also

verifyThat | verifySize | verifyNumElements | length

Purpose

Verify value is less than specified value

Syntax

```
verifyLessThan(verifiable,actual,ceiling)
verifyLessThan(___,diagnostic)
```

Description

`verifyLessThan(verifiable,actual,ceiling)` verifies that all elements of `actual` are less than all the elements of `ceiling`.

`verifyLessThan(___,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThan;
  verifiable.verifyThat(actual, IsLessThan(ceiling));
  ```

  There exists more functionality when using the `IsLessThan` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The value to test. `actual` must be the same size as `ceiling` unless either one is scalar, at which point scalar expansion occurs.

ceiling

Maximum value, exclusive.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Compare Two Numbers

Create a TestCase object for interactive testing.

```
testCase = matlab.unittest.TestCase;
```

Verify 2 is less than 3.

```
testCase.verifyLessThan(2, 3);
```

```
Interactive verification passed.
```

Test if 9 is less than 5.

```
testCase.verifyLessThan(9, 5);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyLessThan failed.
--> The value must be less than the maximum value.

Actual Value:
         9
Maximum Value (Exclusive):
         5
```

Test failed.

Compare an Array to a Scalar

Test if each element is less than the `CEILING` value, `9`.

```
testCase.verifyLessThan([5 6 7], 9);
```

```
Interactive verification passed.
```

Test if each element in the matrix is less than the `CEILING` value, `4`.

```
testCase.verifyLessThan([1 2 3; 4 5 6], 4);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyLessThan failed.
--> Each element must be less than the maximum value.

    Failing Indices:
             2     4     6

Actual Value:
         1     2     3
         4     5     6
Maximum Value (Exclusive):
         4
```

Test failed.

Compare Arrays

Test if each element is less than each corresponding element of the `CEILING` array, `[7 -1 8]`.

```
testCase.verifyLessThan([5 -3 2], [7 -1 8]);
```

```
Interactive verification passed.
```

Compare an array to itself.

```
testCase.verifyLessThan(eye(2), eye(2));
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyLessThan failed.
--> Each element must be less than each corresponding element of the maxi

    Failing Indices:
             1     2     3     4

Actual Value:
         1     0
         0     1
Maximum Value (Exclusive):
         1     0
         0     1
```

Test failed.

See Also

`verifyThat` | `verifyLessThanOrEqual` | `verifyGreaterThan` | `verifyGreaterThanOrEqual` | `lt`

Purpose

Verify value is less than or equal to specified value

Syntax

```
verifyLessThanOrEqual(verifiable,actual,ceiling)
verifyLessThanOrEqual( ___ ,diagnostic)
```

Description

`verifyLessThanOrEqual(verifiable,actual,ceiling)` verifies that all elements of `actual` are less than or equal to all the elements of `ceiling`.

`verifyLessThanOrEqual( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsLessThanOrEqualTo;
  verifiable.verifyThat(actual, IsLessThanOrEqualTo(ceiling));
  ```

  There exists more functionality when using the `IsLessThanOrEqualTo` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The value to test.

ceiling

Maximum value.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Compare Two Numbers

Create a TestCase object for interactive testing.

```
testCase = matlab.unittest.TestCase;
```

Verify 2 is less than 3.

```
testCase.verifyLessThanOrEqual(2, 3);
```

```
Interactive verification passed.
```

Verify 3 is less than or equal to 3.

```
testCase.verifyLessThanOrEqual(3, 3);
```

```
Interactive verification passed.
```

Test if 9 is less than 5.

```
testCase.verifyLessThanOrEqual(9, 5);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyLessThanOrEqual failed.
--> The value must be less than or equal to the maximum value.

Actual Value:
         9
```

```
Maximum Value (Inclusive):
         5
```

Test failed.

Compare an Array to a Scalar

Test if each element is less than or equal to the `CEILING` value, `7`.

```
testCase.verifyLessThanOrEqual([5 2 7], 7);
```

```
Interactive verification passed.
```

Test if each element in the matrix is less than or equal to the `CEILING` value, `4`.

```
testCase.verifyLessThanOrEqual([1 2 3; 4 5 6], 4);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyLessThanOrEqual failed.
--> Each element must be less than or equal to the maximum value.

    Failing Indices:
             4     6

Actual Value:
         1     2     3
         4     5     6
Maximum Value (Inclusive):
         4
```

Test failed.

Compare Arrays

Test if each element is less than or equal to each corresponding element of the CEILING array, [5 -3 8].

```
testCase.verifyLessThanOrEqual([5 -3 2], [5 -3 8]);
```

```
Interactive verification passed.
```

Compare an array to itself.

```
testCase.verifyLessThanOrEqual(eye(2), eye(2));
```

```
Interactive verification passed.
```

See Also

verifyThat | verifyLessThan | verifyGreaterThan | verifyGreaterThanOrEqual | le

Purpose

Verify string matches specified regular expression

Syntax

```
verifyMatches(verifiable,actual,expression)
verifyMatches( ___ ,diagnostic)
```

Description

`verifyMatches(verifiable,actual,expression)` that `actual` is a string that matches the regular expression defined by `expression`.

`verifyMatches( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.Matches;
  verifiable.verifyThat(actual, Matches(expression));
  ```

  There exists more functionality when using the `Matches` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The string to test.

expression

The value to match, specified as a regular expression.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Test for String Matches

Create a TestCase object for interactive testing.

```
testCase = matlab.unittest.TestCase;

testCase.verifyMatches('Some String', 'Some [Ss]tring', ...
    'My result should have matched the expression');

Interactive verification passed.

testCase.verifyMatches('Another string', '(Some |An)other');

Interactive verification passed.

testCase.verifyMatches('Another 3 strings', '^Another \d+ strings?$');

Interactive verification passed.

testCase.verifyMatches('3 more strings', '\d+ strings?');

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyMatches failed.
--> The string did not match the regular expression.

Actual String:
    3 more strings
Regular Expression:
    \d+ strings?
```

Test failed.

See Also

`verifyThat` | `verifySubstring` | `regexp`

matlab.unittest.TestCase.verifyNotEmpty

Purpose

Verify value is not empty

Syntax

```
verifyNotEmpty(verifiable,actual)
verifyNotEmpty(___,diagnostic)
```

Description

`verifyNotEmpty(verifiable,actual)` verifies that `actual` is a non-empty MATLAB value.

`verifyNotEmpty(___,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

```
import matlab.unittest.constraints.IsEmpty;
verifiable.verifyThat(actual, ~IsEmpty());
```

  There exists more functionality when using the `IsEmpty` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The value to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Test for Non-Empty Strings

Create a TestCase object.

```
testCase = matlab.unittest.TestCase;

testCase.verifyNotEmpty('');
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNotEmpty failed.
--> The value must not be empty.
--> The value has a size of [0  0].

Actual Value:
         ''
```

Test failed.

Test for Non-Empty Arrays

An array with any zero dimension is empty.

Test array `[2 3]`.

```
testCase.verifyNotEmpty([2 3]);
```

```
Interactive verification passed.
```

Test array with a zero dimension.

```
testCase.verifyNotEmpty(ones(2, 5, 0, 3));
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNotEmpty failed.
--> The value must not be empty.
--> The value has a size of [2  5  0  3].

Actual Value:
       Empty array: 2-by-5-by-0-by-3
```

Test failed.

Test for Non-Empty Cell Arrays

A cell array of empty arrays is not empty.

```
testCase.verifyNotEmpty({[], [], []}, '');
```

```
Interactive verification passed.
```

Test for Non-Empty Test Suite

Test an empty object, `emptyTestSuite`.

```
emptyTestSuite = matlab.unittest.TestSuite.emtpy;
testCase.verifyNotEmpty(emptyTestSuite);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNotEmpty failed.
--> The value must not be empty.
--> The value has a size of [0  0].

Actual Value:
```

```
0x0 TestCase array with no properties.
```

Test failed.

See Also

`verifyThat` | `verifyEmpty` | `isempty`

matlab.unittest.TestCase.verifyNotEqual

Purpose

Verify value is not equal to specified value

Syntax

```
verifyNotEqual(verifiable,actual,notExpected)
verifyNotEqual( ___ ,diagnostic)
```

Description

`verifyNotEqual(verifiable,actual,notExpected)` verifies that `actual` is not equal to `notExpected`.

`verifyNotEqual( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

```
import matlab.unittest.constraints.IsEqualTo;
verifiable.verifyThat(actual, ~IsEqualTo(notExpected));
```

  There exists more functionality when using the `IsEqualTo` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The value to test.

notExpected

Value to compare.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Compare Numeric Values

Create a TestCase object for interactive testing.

```
testCase = matlab.unittest.TestCase;
```

Compare a value to itself.

```
testCase.verifyNotEqual(5, 5);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNotEqual failed.
--> NumericComparator passed.

Actual Value:
         5
Prohibited Value:
         5
```

Test failed.

Compare different number values.

```
testCase.verifyNotEqual(4.95, 5, '4.95 should be different from 5');
```

```
Interactive verification passed.
```

Values `4.95` and `5` are not equal.

Compare values of different sizes.

```
testCase.verifyNotEqual([5 5], 5, '[5 5] is not equal to 5');
```

```
Interactive verification passed.
```

Values are not equal.

Compare Classes

```
testCase.verifyNotEqual(int8(5), int16(5), 'Classes dont match');
```

```
Interactive verification passed.
```

Compare Cell Arrays

Test a cell array by comparing each element.

```
testCase.verifyNotEqual({'cell', struct, 5}, {'cell', struct, 5});
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNotEqual failed.
--> CellComparator passed.

Actual Value:
        'cell'    [1x1 struct]    [5]
Prohibited Value:
        'cell'    [1x1 struct]    [5]
```

Test failed.

See Also

verifyThat | verifyEqual |

Purpose

Verify value is not handle to specified instance

Syntax

```
verifyNotSameHandle(verifiable,actual,notExpectedHandle)
verifyNotSameHandle( ___ ,diagnostic)
```

Description

`verifyNotSameHandle(verifiable,actual,notExpectedHandle)` verifies that `actual` is a different size and/or does not contain the same instances as the `notExpectedHandle` handle array.

`verifyNotSameHandle( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  verifiable.verifyThat(actual, ~IsSameHandleAs(notExpectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The value to test.

notExpectedHandle

The handle array to compare.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Test Handles from Same Class

Create a TestCase object for interactive testing.

```
testCase = matlab.unittest.TestCase;
```

Create a handle class, ExampleHandle.

```
classdef ExampleHandle < handle
end
```

Create two handle variables.

```
h1 = ExampleHandle;
h2 = ExampleHandle;
```

Handles point to different objects.

```
testCase.verifyNotSameHandle(h1, h2);
```

```
Interactive verification passed.
```

Show matching handle combinations.

```
testCase.verifyNotSameHandle([h1 h2 h1], [h1 h2 h1]);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNotSameHandle failed.
```

```
--> The two handles must not refer to the same handle, or should have
different sizes.

Actual Value:
      1x3 ExampleHandle array with no properties.
Handle Object:
      1x3 ExampleHandle array with no properties.
```

Test failed.

The order of the handle arguments matters.

```
testCase.verifyNotSameHandle([h1 h2], [h2 h1]);
```

```
Interactive verification passed.
```

Test a handle with itself.

```
testCase.verifyNotSameHandle(h1, h1);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNotSameHandle failed.
--> The two handles must not refer to the same handle, or should have
different sizes.

Actual Value:
      ExampleHandle with no properties.
Handle Object:
      ExampleHandle with no properties.
```

Test failed.

Variables are not same size.

```
testCase.verifyNotSameHandle(h2, [h2 h2]);
```

```
Interactive verification passed.
```

Variables are the same size.

```
testCase.verifyNotSameHandle([h1 h1], [h1 h1]);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNotSameHandle failed.
--> The two handles must not refer to the same handle, or should have
different sizes.

Actual Value:
      1x2 ExampleHandle array with no properties.
Handle Object:
      1x2 ExampleHandle array with no properties.
```

Test failed.

See Also

verifySameHandle | verifyThat |

Purpose

Verify value has specified element count

Syntax

```
verifyNumElements(verifiable,actual,expectedElementCount)
verifyNumElements( ___ ,diagnostic)
```

Description

`verifyNumElements(verifiable,actual,expectedElementCount)` verifies that `actual` is a MATLAB array with `expectedElementCount` number of elements.

`verifyNumElements( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasElementCount;
  verifiable.verifyThat(actual, HasElementCount(expectedElementCount)
  ```

  There exists more functionality when using the `HasElementCount` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The value to test.

expectedElementCount

The expected number of elements in the array.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Test Matrices

Create a TestCase object for interactive testing.

```
testCase = matlab.unittest.TestCase;

n = 7;
testCase.verifyNumElements(eye(n), n^2);

Interactive verification passed.

testCase.verifyNumElements(3, 1);

Interactive verification passed.

testCase.verifyNumElements([1 2 3; 4 5 6], 5);

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNumElements failed.
--> The value did not have the correct number of elements.

    Actual Number of Elements:
             6
    Expected Number of Elements:
             5

Actual Value:
```

```
     1     2     3
     4     5     6
```

Test failed.

Test Cell Array

```
testCase.verifyNumElements({'SomeString', 'SomeOtherString'}, 2);
```

```
Interactive verification passed.
```

Test Structure

```
s.Field1 = 1;
s.Field2 = 2;
testCase.verifyNumElements(s, 2);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyNumElements failed.
--> The value did not have the correct number of elements.

    Actual Number of Elements:
             1
    Expected Number of Elements:
             2

Actual Value:
        Field1: 1
        Field2: 2
```

Test failed.

See Also

verifyThat | verifySize | verifyLength | numel

matlab.unittest.TestCase.verifyReturnsTrue

Purpose

Verify function returns true when evaluated

Syntax

```
verifyReturnsTrue(verifiable,actual)
verifyReturnsTrue(___,diagnostic)
```

Description

`verifyReturnsTrue(verifiable,actual)` verifies that `actual` is a function handle that returns a scalar logical whose value is true.

`verifyReturnsTrue(___,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- It is a shortcut for quick custom comparison functionality that can be defined quickly, and possibly inline. It can be preferable over simply evaluating the function directly and using verifyTrue because the function handle will be shown in the diagnostics, thus providing more insight into the failure condition which is lost when using verifyTrue.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ReturnsTrue;
  verifiable.verifyThat(actual, ReturnsTrue());
  ```

  There exists more functionality when using the `ReturnsTrue` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The function handle to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Test if Condition is True

Create a TestCase object for interactive testing.

```
testCase = matlab.unittest.TestCase;

testCase.verifyReturnsTrue(@true);

Interactive verification passed.

testCase.verifyReturnsTrue(@() isequal(1,1));

Interactive verification passed.

testCase.verifyReturnsTrue(@() ~strcmp('a','b'));

Interactive verification passed.

testCase.verifyReturnsTrue(@false);

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyReturnsTrue failed.
--> The function handle should have evaluated to "true".
--> Returned value:
             0
```

```
Actual Function Handle:
        @false
```

Test failed.

```
testCase.verifyReturnsTrue(@() strcmp('a',{'a','a'}));
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyReturnsTrue failed.
--> The function handle should have returned a scalar. The return value h
--> Returned value:
             1     1

Actual Function Handle:
        @()strcmp('a',{'a','a'})
```

Test failed.

```
testCase.verifyReturnsTrue(@() exist('exist'));
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyReturnsTrue failed.
--> The function handle should have returned a logical value. It was of t
--> Returned value:
             5

Actual Function Handle:
        @()exist('exist')
```

Test failed.

See Also

`verifyThat` | `verifyTrue` |

matlab.unittest.TestCase.verifySameHandle

Purpose

Verify two values are handles to same instance

Syntax

```
verifySameHandle(verifiable,actual,expectedHandle)
verifySameHandle( ___ ,diagnostic)
```

Description

`verifySameHandle(verifiable,actual,expectedHandle)` verifies that `actual` is the same size and contains the same instances as the `expectedHandle` handle array.

`verifySameHandle( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsSameHandleAs;
  verifiable.verifyThat(actual, IsSameHandleAs(expectedHandle));
  ```

  There exists more functionality when using the `IsSameHandleAs` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The value to test.

expectedHandle

The expected handle array.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Test Handles from Same Class

Create a TestCase object for interactive testing.

```
testCase = matlab.unittest.TestCase;
```

Create a handle class, `ExampleHandle`.

```
classdef ExampleHandle < handle
end
```

Create two handle variables.

```
h1 = ExampleHandle;
h2 = ExampleHandle;
```

Show matching handle combinations.

```
testCase.verifySameHandle(h1, h1);
```

```
Interactive verification passed.
```

```
testCase.verifySameHandle([h1 h1], [h1 h1]);
```

```
Interactive verification passed.
```

```
testCase.verifySameHandle([h1 h2 h1], [h1 h2 h1]);
```

```
Interactive verification passed.
```

Handles must point to same object.

```
testCase.verifySameHandle(h1, h2);

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySameHandle failed.
--> Values do not refer to the same handle.

Actual Value:
      ExampleHandle with no properties.
Expected Handle Object:
      ExampleHandle with no properties.
```

Test failed.

Size of handle objects must match.

```
testCase.verifySameHandle([h1 h1], h1);

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySameHandle failed.
--> Sizes do not match.
     Actual Value Size            : [1  2]
     Expected Handle Object Size : [1  1]

Actual Value:
      1x2 ExampleHandle array with no properties.
Expected Handle Object:
      ExampleHandle with no properties.
```

Test failed.

Order of arguments is important.

```
testCase.verifySameHandle([h1 h2], [h2 h1]);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySameHandle failed.
--> Some elements in the handle array refer to the wrong handle.

Actual Value:
      1x2 ExampleHandle array with no properties.
Expected Handle Object:
      1x2 ExampleHandle array with no properties.
```

Test failed.

See Also

verifyThat | verifyNotSameHandle | handle

matlab.unittest.TestCase.verifySize

Purpose

Verify value has specified size

Syntax

```
verifySize(verifiable,actual,expectedSize)
verifySize( ___ ,diagnostic)
```

Description

`verifySize(verifiable,actual,expectedSize)` verifies that `actual` is a MATLAB array whose size is `expectedSize`.

`verifySize( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.HasSize;
  verifiable.verifyThat(actual, HasSize(expectedSize));
  ```

  There exists more functionality when using the `HasSize` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The value to test.

expectedSize

The expected sizes of each dimension the array.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Test Arrays

Create a TestCase object for interactive testing.

```
testCase = matlab.unittest.TestCase;
```

```
testCase.verifySize(ones(2, 5, 3), [2 5 3]);
```

```
Interactive verification passed.
```

```
testCase.verifySize([1 2 3; 4 5 6], [2 3]);
```

```
Interactive verification passed.
```

```
testCase.verifySize([2 3], [3 2]);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySize failed.
--> The value had an incorrect size.

    Actual Size:
             1     2
    Expected Size:
             3     2

Actual Value:
         2     3
```

Test failed.

Number of elements is not the same as size.

```
testCase.verifySize([1 2 3; 4 5 6], [6 1]);

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySize failed.
--> The value had an incorrect size.

    Actual Size:
             2     3
    Expected Size:
             6     1

Actual Value:
         1     2     3
         4     5     6
```

Test failed.

```
testCase.verifySize(eye(2), [4 1]);

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySize failed.
--> The value had an incorrect size.

    Actual Size:
             2     2
    Expected Size:
             4     1
```

```
Actual Value:
         1     0
         0     1
```

Test failed.

Test Cell Array

```
testCase.verifySize({'SomeString', 'SomeOtherString'}, [1 2]);
```

```
Interactive verification passed.
```

See Also

verifyThat | verifyLength | verifyNumElements | size

matlab.unittest.TestCase.verifySubstring

Purpose

Verify string contains specified string

Syntax

```
verifySubstring(verifiable,actual,substring)
verifySubstring( ___ ,diagnostic)
```

Description

`verifySubstring(verifiable,actual,substring)` verifies that `actual` is a string that contains `substring`.

`verifySubstring( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.ContainsSubstring;
  verifiable.verifyThat(actual, ContainsSubstring(substring));
  ```

  There exists more functionality when using the `ContainsSubstring` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The string to test.

substring

The value to match, specified as a string.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle

- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Test for Substrings

Create a TestCase object for interactive testing.

```
testCase = matlab.unittest.TestCase;

testCase.verifySubstring('SomeLongString', 'Long');

Interactive verification passed.
```

Show that case matters.

```
testCase.verifySubstring('SomeLongString', 'lonG');

Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySubstring failed.
--> The string must contain the substring.

Actual String:
    SomeLongString
Expected Substring:
    lonG
```

Test failed.

```
testCase.verifySubstring('SomeLongString', 'OtherString');

Interactive verification failed.
```

```
---------------------
Framework Diagnostic:
---------------------
verifySubstring failed.
--> The string must contain the substring.

Actual String:
    SomeLongString
Expected Substring:
    OtherString
```

Test failed.

```
testCase.verifySubstring('SomeLongString', 'SomeLongStringThatIsLonger');
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifySubstring failed.
--> The string must contain the substring.

Actual String:
    SomeLongString
Expected Substring:
    SomeLongStringThatIsLonger
```

Test failed.

See Also

verifyThat | verifyMatches | strfind

Purpose

Verify value meets given constraint

Syntax

```
verifyThat(verifiable,actual,constraint)
verifyThat( ___ ,diagnostic)
```

Description

`verifyThat(verifiable,actual,constraint)` verifies that `actual` is a value that satisfies the `constraint` provided.

If the constraint is not satisfied, a verification failure is produced utilizing only the framework diagnostic generated by the `constraint`.

`verifyThat( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

When using this signature, both the diagnostic information contained within `diagnostic` is used in addition to the diagnostic information provided by the `constraint`.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The value to test.

constraint

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Test Conditions Using Constraints

```
% Passing scenarios
%%%%%%%%%%%%%%%%%%%%
import matlab.unittest.constraints.IsTrue;
testCase.verifyThat(true, IsTrue);

import matlab.unittest.constraints.IsEqualTo;
testCase.verifyThat(5, IsEqualTo(5), '5 should be equal to 5');

import matlab.unittest.constraints.IsGreaterThan;
import matlab.unittest.constraints.HasNaN;
testCase.verifyThat([5 NaN], IsGreaterThan(10) | HasNaN, ...
    'The value was not greater than 10 or NaN');


% Failing scenarios
%%%%%%%%%%%%%%%%%%%%
import matlab.unittest.constraints.AnyCellOf;
import matlab.unittest.constraints.ContainsSubstring;
testCase.verifyThat( AnyCellOf({'cell','of','strings'}), ...
    ContainsSubstring('char'),'Test description');

import matlab.unittest.constraints.HasSize;
testCase.verifyThat(zeros(10,4,2), HasSize([10,5,2]), ...
    @() disp('A function handle diagnostic.'));

import matlab.unittest.constraints.IsEmpty;
testCase.verifyThat(5, IsEmpty);
```

Purpose

Verify value is true

Syntax

```
verifyTrue(verifiable,actual)
verifyTrue(___,diagnostic)
```

Description

`verifyTrue(verifiable,actual)` verifies that `actual` is a scalar logical with the value of true.

`verifyTrue(___,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

Tips

- This method passes if and only if the actual value is a scalar logical with a value of true. Therefore, entities such as true valued arrays and nonzero doubles produce qualification failures when used in this method, despite these entities exhibiting "true-like" behavior such as triggering the execution of code inside of "if" statements.
- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IsTrue;
  verifiable.verifyThat(actual, IsTrue());
  ```

  There exists more functionality when using the `IsTrue` constraint directly via verifyThat.

  Use of this method for performance benefits can come at the expense of less diagnostic information, and may not provide the same level of strictness adhered to by other constraints such as IsEqualTo. A similar approach that is generally less performant but can provide slightly better diagnostic information is the use of verifyReturnsTrue, which at least shows the display of the function evaluated to generate the failing result.
-

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The value to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Examples

Test MATLAB Logical Functions

Create a TestCase object for interactive testing.

```
testCase = matlab.unittest.TestCase;
```

Test `true`.

```
testCase.verifyTrue(true);
```

```
Interactive verification passed.
```

Test `false`.

```
testCase.verifyTrue(false);
```

```
Interactive verification failed.
```

```
---------------------
Framework Diagnostic:
---------------------
verifyTrue failed.
--> The value must evaluate to "true".

Actual Value:
         0
```

Test failed.

Test the Value 1

The number `1` is a `double` value, not a logical value.

A double value of `1` is not `true`.

```
testCase.verifyTrue(1);
```

```
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyTrue failed.
--> The value must be logical. It is of type "double".

Actual Value:
         1
```

Test failed.

Test Array of Logical Values

To be true, the value must be scalar.

```
testCase.verifyTrue([true true true]);
```

```
Interactive verification failed.
```

```
---------------------
Framework Diagnostic:
---------------------
verifyTrue failed.
--> The value must be scalar. It has a size of [1  3].

Actual Value:
         1     1     1
```

Test failed.

Optimized comparison that trades speed for less diagnostics.

```
testCase.verifyTrue(~isempty(strfind('string', 'ring')), ...
    'Could not find expected string');
```

See Also

verifyThat | verifyFalse | verifyReturnsTrue |

matlab.unittest.TestCase.verifyWarning

Purpose

Verify function issues specified warning

Syntax

```
verifyWarning(verifiable,actual,warningID)
verifyWarning(___,diagnostic)
[output1,...,outputN] = verifyWarning(___)
```

Description

`verifyWarning(verifiable,actual,warningID)` verifies that `actual` issues a warning with the identifier `warningID`.

`verifyWarning(___,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`[output1,...,outputN] = verifyWarning(___)` also returns the output arguments `output1,...,outputN` that are produced when invoking `actual`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesWarnings;
  verifiable.verifyThat(actual, IssuesWarnings({warningID}));
  ```

  There exists more functionality when using the `IssuesWarnings` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The function handle to test.

warningID

Warning ID, specified as a string.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string
- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Output Arguments

output1,...,outputN

Output arguments, 1 through n (if any), from `actual`, returned as any type. The argument type is specified by the `actual` argument list.

Examples

Test warning Function

Create a TestCase object for interactive testing.

```
testCase = matlab.unittest.TestCase;

testCase.verifyWarning(@() warning('SOME:warning:id','Warning!'), ...
    'SOME:warning:id');

Interactive verification passed.

testCase.verifyWarning(@() warning('SOME:other:id', 'Warning message'),..
    'SOME:warning:id', 'Did not issue specified warning');

Warning: Warning message
> In @()warning('SOME:other:id','Warning message')
  In FunctionHandleConstraint>FunctionHandleConstraint.invoke at 43
  In WarningQualificationConstraint>WarningQualificationConstraint.invoke
  In IssuesWarnings>IssuesWarnings.invoke at 364
  In IssuesWarnings>IssuesWarnings.issuesExpectedWarnings at 411
  In IssuesWarnings>IssuesWarnings.satisfiedBy at 240
```

```
  In QualificationDelegate>QualificationDelegate.qualifyThat at 90
  In QualificationDelegate>QualificationDelegate.qualifyWarning at 196
  In Verifiable>Verifiable.verifyWarning at 701
Interactive verification failed.

----------------
Test Diagnostic:
----------------
Did not issue specified warning

---------------------
Framework Diagnostic:
---------------------
verifyWarning failed.
--> The function handle did not issue a correct warning profile.
    The expected warning profile ignores:
      Set
      Count
      Order
    --> The function handle did not issue the correct warnings.

        Missing Warnings:
             SOME:warning:id

    Actual Warning Profile:
         SOME:other:id
    Expected Warning Profile:
         SOME:warning:id

Evaluated Function:
        @()warning('SOME:other:id','Warning message')
```

Test a Function Without Warnings

Test the `true` function, which does not issue warnings.

```
testCase.verifyWarning(@true, 'SOME:warning:id', ...
```

```
'@true did not issue any warning');

Interactive verification failed.

----------------
Test Diagnostic:
----------------
@true did not issue any warning

---------------------
Framework Diagnostic:
---------------------
verifyWarning failed.
--> The function handle did not issue a correct warning profile.
    The expected warning profile ignores:
      Set
      Count
      Order
    --> The function handle did not issue any warnings.

    Expected Warning Profile:
         SOME:warning:id

Evaluated Function:
        @true
```

Test failed.

Test Function With Output Arguments

Create a `helper` function that generates a warning and returns output.

```
function varargout = helper()
    warning('SOME:warning:id','Warning!');
    varargout = {123, 'abc'};
end
```

Call `helper`.

```
[actualOut1, actualOut2] = testCase.verifyWarning(@helper, ...
    'SOME:warning:id');

Interactive verification passed.
```

See Also

verifyThat | verifyError | verifyWarningFreewarning |

matlab.unittest.TestCase.verifyWarningFree

Purpose

Verify function issues no warnings

Syntax

```
verifyWarningFree(verifiable,actual)
verifyWarningFree( ___ ,diagnostic)
output1,...,outputN = verifyWarningFree( ___ )
```

Description

`verifyWarningFree(verifiable,actual)` verifies that `actual` is a function handle that issues no warnings.

`verifyWarningFree( ___ ,diagnostic)` also displays the diagnostic information in `diagnostic` upon a failure.

`output1,...,outputN = verifyWarningFree( ___ )` also returns the output arguments `output1,...,outputN` that are produced when invoking `actual`.

Tips

- This method is functionally equivalent to:

  ```
  import matlab.unittest.constraints.IssuesNoWarnings;
  verifiable.verifyThat(actual, IssuesNoWarnings());
  ```

  There exists more functionality when using the `IssuesNoWarnings` constraint directly via verifyThat.

Input Arguments

verifiable

The matlab.unittest.TestCase instance which is used to pass or fail the verification in conjunction with the test running framework.

actual

The function handle to test.

diagnostic

Diagnostic information to display upon a failure, specified as one of the following:

- string

- function handle
- matlab.unittest.diagnostics.Diagnostic object

Diagnostic values can be nonscalar. For more information, see matlab.unittest.diagnostics.Diagnostic.

Output Arguments

output1,...,outputN

Output arguments, 1 through n (if any), from `actual`, returned as any type. The argument type is specified by the `actual` argument list.

Examples

Test for Warnings from MATLAB Functions

Create a TestCase object for interactive testing.

```
testCase = matlab.unittest.TestCase;
```

Test the `why` function.

```
testCase.verifyWarningFree(@why);
```

```
The bald and not excessively bald and not excessively smart hamster ob
Interactive verification passed.
```

This is a randomly-generated message.

Test the `true` function.

```
testCase.verifyWarningFree(@true);
```

```
Interactive verification passed.
```

Test the `false` function.

```
actualOutputFromFalse = testCase.verifyWarningFree(@false);
```

```
Interactive verification passed.
```

Test a value that is not a function handle.

```
testCase.verifyWarningFree(5,'diagnostic');

Interactive verification failed.

----------------
Test Diagnostic:
----------------
diagnostic

---------------------
Framework Diagnostic:
---------------------
verifyWarningFree failed.
--> The value must be an instance of the expected type.

    Actual Class:
        double
    Expected Type:
        function_handle

Actual Value:
         5
```

Test failed.

Test a function that generates warning.

```
testCase.verifyWarningFree(@() warning('some:id', 'Message'));

Warning: Message
> In @()warning('some:id','Message')
  In FunctionHandleConstraint>FunctionHandleConstraint.invoke at 43
  In WarningQualificationConstraint>WarningQualificationConstraint.invoke
  In IssuesNoWarnings>IssuesNoWarnings.issuesNoWarnings at 131
  In IssuesNoWarnings>IssuesNoWarnings.satisfiedBy at 82
  In QualificationDelegate>QualificationDelegate.qualifyThat at 90
```

```
  In QualificationDelegate>QualificationDelegate.qualifyWarningFree at
  In Verifiable>Verifiable.verifyWarningFree at 757
Interactive verification failed.

---------------------
Framework Diagnostic:
---------------------
verifyWarningFree failed.
--> The function issued warnings.

    Warnings Issued:
         some:id

Evaluated Function:
        @()warning('some:id','Message')
```

Test failed.

See Also

verifyThat | verifyWarning | warning

Purpose

Result of running test suite

Description

The matlab.unittest.TestResult class holds the information describing the result of running a test suite using the matlab.unittest framework. The results include information describing whether the test passed, failed, or ran to completion, as well as the duration of each test.

Construction

TestResult arrays are created and returned by the test runner, and are of the same size as the suite which was run.

Properties

Duration

Time elapsed running test.

The `Duration` property indicates the amount of time taken to run a particular test, including the time taken setting up and tearing down any test fixtures.

Fixture setup time is accounted for in the duration of the first test suite array element that uses the fixture. Fixture teardown time is accounted for in the duration of the last test suite array element that uses the fixture.

The total run time for a suite of tests exceeds the sum of the durations for all the elements of the suite because the `Duration` property does not include all the overhead of the TestRunner object, nor any of the time consumed by test runner plugins.

Failed

Logical value showing if test failed.

A TRUE `Failed` property indicates some form of test failure. When `Failed` is FALSE, then no failing conditions were encountered. A failing result can occur with a failure condition either in a test or in setting up and tearing down test fixtures. Failures can occur due to the following:

- Verification failures

- Assertion failures
- Uncaught MExceptions

Fatal assertions are also failing conditions, but in the event of a fatal assertion failure, the entire framework aborts and a TestResult object is never produced.

Incomplete

Logical value showing if test did not run to completion.

A TRUE `Incomplete` property indicates a test did not run to completion. When it is FALSE, then no conditions were encountered that prevented the test from completing. In other words, when FALSE there were no stack disruptions out of the running test content. An incomplete result can occur with a stack disruption in either a test or when setting up and tearing down test fixtures. Incomplete tests can occur due to the following:

- Assertion failures
- Tests filtered through assumption
- Uncaught MExceptions

Fatal assertions are also conditions that prevent the completion of tests, but in the event of a fatal assertion failure the entire framework aborts and a TestResult object is never produced.

Name

The name of the TestSuite object for the result.

The `Name` property is a string that holds the name of the test corresponding to this result.

Passed

Logical value showing if the test passed.

When the `Passed` property is TRUE, then the test completed as expected without any failure. When it is FALSE, then the test did not run to completion and/or encountered a failure condition.

Examples

Use TestResult Object to Identify and Rerun Failed Tests

Add the matlab.unittest.TestSuite class to the current import list.

```
import matlab.unittest.TestSuite;
```

Result display method provides a summary of the results

```
suite = TestSuite.fromClass(?SomeTestClass)
results = run(suite)
```

```
results =

  matlab.unittest.TestResult

  Test Suite Summary:
    12 Passed, 4 Failed, 0 Not Run To Completion.
    5.5091 seconds testing time.
```

Rerun only the failed tests.

```
failedTests = suite([results.Failed]);
failedResults = run(failedTests)
```

```
failedResults =

  matlab.unittest.TestResult

  Test Suite Summary:
    0 Passed, 4 Failed, 0 Not Run To Completion.
    1.2894 seconds testing time.
```

Make the fix and rerun results.

```
newResults = run(failedTests)

newResults =

  matlab.unittest.TestResult

  Test Suite Summary:
    4 Passed, 0 Failed, 0 Not Run To Completion.
    1.1607 seconds testing time.
```

See Also

`TestRunner` | `TestSuite` |

Concepts

- Property Attributes

matlab.unittest.TestRunner

Purpose

Class for running tests in `matlab.unittest` framework

Description

The matlab.unittest.TestRunner class is the fundamental API used to run a suite of tests in the matlab.unittest framework. It runs and operates on TestSuite arrays. Use this class to customize running tests.

The TestRunner class is a sealed class; you cannot derive classes from the TestRunner class.

Construction

To create a simple, silent TestRunner object, call the static withNoPlugins method.

```
runner = matlab.unittest.TestRunner.withNoPlugins
```

To create a TestRunner object to run tests from the MATLAB Command Window , call the static withTextOutput method.

```
runner = matlab.unittest.TestRunner.withTextOutput
```

To create a customized TestRunner object, call the addPlugin method.

```
runner = TestRunner.withNoPlugins;
runner.addPlugin(SomePlugin())
```

Methods

addPlugin	Add plugin to `TestRunner` object
run	Run all tests in `TestSuite` array
withNoPlugins	Create simplest runner possible
withTextOutput	Create `TestRunner` object for command window output

Examples

Create TestRunner Object Configured for Text Output

Add matlab.unittest classes to the current import list.

```
import matlab.unittest.TestRunner;
```

```
import matlab.unittest.TestSuite;
```

Create a TestSuite array.

```
suite = TestSuite.fromClass(?mypackage.MyTestClass);
```

Create the TestRunner object and run the suite.

```
runner = TestRunner.withTextOutput;
result = run(runner,suite);
```

See Also

TestSuite | TestResult |

matlab.unittest.TestRunner.addPlugin

Purpose

Add plugin to `TestRunner` object

Syntax

`addPlugin(runner,plugin)`

Description

`addPlugin(runner,plugin)` adds `plugin` to `runner`.

Input Arguments

runner

matlab.unittest.TestRunner object.

plugin

Mechanism provided to customize the manner in which a TestSuite array is run, specified as a TestRunnerPlugin object.

Examples

Run Test with Custom Plugin

Add matlab.unittest classes to the current import list.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
```

Create a TestSuite array.

```
suite = TestSuite.fromClass(?mypackage.MyTestClass);
```

Create a TestRunner object.

```
runner = TestRunner.withNoPlugins;
```

Add a custom plugin.

```
import matlab.unittest.plugins;
runner.addPlugin(DiagnosticsValidationPlugin);
```

Run the test.

```
result = run(runner,suite);
```

Purpose Run all tests in `TestSuite` array

Syntax

```
result = run(runner,suite)
```

Description

`result = run(runner,suite)` runs the TestSuite array defined by `suite` using the TestRunner object provided in `runner`, and returns the result in `result`.

This method runs all of the appropriate methods of the TestCase class to set up fixtures and run test content. It handles errors and qualification failures and records the information in `result`.

Input Arguments

runner

matlab.unittest.TestRunner object.

suite

Set of tests, specified as a matlab.unittest.TestSuite array.

Output Arguments

result

A matlab.unittest.TestResult object containing the result of the test run. `result` is the same size as `suite` and each element is the result of the corresponding element in `suite`.

Examples

Run All Tests in a Package

Add matlab.unittest classes to the current import list.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
```

Create a test suite, and a test runner that displays text.

```
suite = TestSuite.fromClass(?mypackage.MyTestClass);
runner = TestRunner.withTextOutput;
```

Run the test suite.

```
result = runner.run(suite)
```

See Also

matlab.unittest.TestSuite.run | matlab.unittest.TestCase.run |

Purpose Create simplest runner possible

Syntax

```
runner = matlab.unittest.TestRunner.withNoPlugins
```

Description

`runner = matlab.unittest.TestRunner.withNoPlugins` creates a TestRunner that is guaranteed to have no plugins installed and returns it in `runner`. It is the method one can use to create the simplest runner possible without violating the guarantees a test writer has when writing TestCase classes. This runner is a silent runner, meaning that regardless of passing or failing tests, this runner produces no command window output, although the results returned after running a test suite are accurate.

This method can also be used when it is desirable to have complete control over which plugins are installed and in what order. It is the only method guaranteed to produce the minimal TestRunner with no plugins, so one can create it and add additional plugins as desired.

Output Arguments

runner

matlab.unittest.TestRunner object.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Create a Silent TestRunner Object with no Plugins

Add matlab.unittest classes to the current import list.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
```

Create a TestSuite array.

```
suite = TestSuite.fromClass(?mypackage.MyTestClass);
```

Create a TestRunner object.

```
runner = TestRunner.withNoPlugins;

% Run the suite silently
result = run(runner,suite)
```

Control Plugins

Using the TestRunner object created in the previous example, control which plugins are installed and in what order they are installed.

Add matlab.unittest class to the current import list.

```
import matlab.unittest.plugins;
```

Add specific plugins.

```
runner.addPlugin(DiagnosticsValidationPlugin);
runner.addPlugin(TestSuiteProgressPlugin);
```

Rerun the tests.

```
result = run(runner,suite)
```

Purpose

Create `TestRunner` object for command window output

Syntax

`runner = matlab.unittest.TestRunner.withTextOutput`

Description

`runner = matlab.unittest.TestRunner.withTextOutput` creates a TestRunner object that is configured for running tests from the MATLAB Command Window and returns it in `runner`. The output produced includes test progress as well as diagnostics in the event of test failures.

Output Arguments

runner

matlab.unittest.TestRunner object.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Display Test Results in Command Window

Add matlab.unittest classes to the current import list.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
```

Create a TestSuite array.

```
suite = TestSuite.fromClass(?mypackage.MyTestClass);
```

Create a TestRunner object that produced output to the Command Window.

```
runner = TestRunner.withTextOutput;
```

```
% Run the suite
result = run(runner,suite)
```

See Also run |

Purpose

Class for grouping tests to run

Description

The matlab.unittest.TestSuite class is the fundamental interface used to group and run a set of tests in the unit test framework. The matlab.unittest.TestRunner object can only run arrays of TestSuite objects.

Construction

TestSuite arrays are created using static methods of the TestSuite class. These methods may return subclasses of the TestSuite class depending on the method call and context.

Methods

fromClass	Create suite from `TestCase` class
fromFile	Create `TestSuite` array from test file
fromFolder	Create `TestSuite` array from all tests in folder
fromMethod	Create `TestSuite` array from single test method
fromPackage	Create `TestSuite` array from all tests in package
run	Run `TestSuite` array using `TestRunner` object configured for text output

Examples

Create Test Suite of Every Type of Test Set

Add the matlab.unittest.TestSuite class to the current import list.

```
import matlab.unittest.TestSuite;
```

Create test suites using a each method.

```
fileSuite    = TestSuite.fromFile('SomeTestFile.m');
```

```
folderSuite  = TestSuite.fromFolder(pwd);
packageSuite = TestSuite.fromPackage('mypackage.subpackage');
classSuite   = TestSuite.fromClass(?mypackage.MyTestClass);
methodSuite  = TestSuite.fromMethod(?SomeTestClass,'testMethod');
```

Concatenate the suites.

```
largeSuite = [fileSuite, folderSuite, packageSuite, classSuite, methodSui
```

Run the full suite.

```
result = run(largeSuite)
```

See Also

TestRunner | TestResult |

Purpose

Create suite from `TestCase` class

Syntax

```
suite = matlab.unittest.TestSuite.fromClass(testClass)
```

Description

`suite = matlab.unittest.TestSuite.fromClass(testClass)` creates a TestSuite array from all of the `Test` methods contained in `testClass` and returns that array in `suite`.

Tips

- `testClass` must be on the MATLAB path when using this method to create `suite`, as well as when `suite` is run.

Input Arguments

testClass

Class containing test methods, specified as a `meta.class` instance. Use the `?` operator to create a `meta.class` instance. `testClass` must derive from matlab.unittest.TestCase.

Output Arguments

suite

Set of tests, specified as a matlab.unittest.TestSuite array.

Attributes

Static	true

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Run Tests in a Package Class

Add the matlab.unittest.TestSuite class to the current import list.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromClass(?mypackage.MyTestClass);
result = run(suite)
```

Run Tests in a Class Without a Package

Add the matlab.unittest.TestSuite class to the current import list.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromClass(?MyTestClass);
result = run(suite)
```

See Also

TestRunner | fromMethod | fromPackage |

Purpose

Create `TestSuite` array from test file

Syntax

```
suite = matlab.unittest.TestSuite.fromFile(file)
```

Description

`suite = matlab.unittest.TestSuite.fromFile(file)` creates a TestSuite array from all of the `Test` methods in `file`.

When the test suite is run, MATLAB changes the current folder to the folder the suite was created from, and adds it to the path for the duration of the test run.

Input Arguments

file

Class file derived from matlab.unittest.TestCase containing test methods, specified as a string.

Output Arguments

suite

Set of tests, specified as a matlab.unittest.TestSuite array.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Run Tests in Class File

Function for unit testing:

```
function res = add5(x)
%ADD5 Increment input by 5.
if ~isa(x,'numeric')
    error('add5:InputMustBeNumeric','Input must be numeric.');
end
res = x + 5;
end
```

TestCase class containing test methods:

```
classdef Add5Test < matlab.unittest.TestCase
    methods (Test)
        function testDoubleOut(testCase)
            actOutput = add5(1);
            testCase.verifyClass(actOutput,'double');
        end
        function testNonNumericInput(testCase)
            testCase.verifyError(@()add5('0'),'add5:InputMustBeNumeric');
        end
    end
end
```

Create a test suite from the `Add5Test` class file.

```
suite = matlab.unittest.TestSuite.fromFile('Add5Test.m');

result = run(suite);
```

```
Running Add5Test
..
Done Add5Test
__________
```

See Also

TestRunner | fromFolder |

Purpose

Create `TestSuite` array from all tests in folder

Syntax

```
suite = matlab.unittest.TestSuite.fromFolder(folder)
suite = matlab.unittest.TestSuite.fromFolder(folder,Name,
   Value)
```

Description

`suite = matlab.unittest.TestSuite.fromFolder(folder)` creates a TestSuite array from all of the `Test` methods of all concrete TestCase classes contained in `folder` and returns that array in `suite`. The method is not recursive, returning only those tests directly in the specified folder.

When the test suite is run, MATLAB changes the current folder to the folder the suite was created from, and adds it to the path for the duration of the test run.

`suite = matlab.unittest.TestSuite.fromFolder(folder,Name,Value)` creates a TestSuite array with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

folder

Folder containing test methods, specified as a string. `folder` can be either an absolute or relative path to the desired folder.

Name-Value Pair Arguments

'IncludingSubfolders'

Include test methods from any `folder` subfolders, excluding package, class, and private folders.

Output Arguments

suite

Set of tests, specified as a matlab.unittest.TestSuite array.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Run Tests in Current Folder

Add the matlab.unittest.TestSuite class to the current import list.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromFolder(pwd);
result = run(suite);
```

Run Tests in Subfolders

```
suite = TestSuite.fromFolder(pwd, 'IncludingSubfolders', true);
result = run(suite);
```

See Also

TestRunner | fromFile |

Purpose

Create `TestSuite` array from single test method

Syntax

```
suite = matlab.unittest.TestSuite.fromMethod(testClass,
   testMethod)
```

Description

`suite =`
`matlab.unittest.TestSuite.fromMethod(testClass,testMethod)` creates a TestSuite array from the test class described by `testClass` and the test method described by `testMethod` and returns it in `suite`.

Tips

- `testClass` must be on the MATLAB path when using this method to create `suite`, as well as when `suite` is run.

Input Arguments

testClass

Class describing the test methods, specified as a `meta.class` instance which must derive from matlab.unittest.TestCase.

testMethod

Test method, specified by either the `meta.method` instance or the name as a string. The method must be defined with a `Test` method attribute.

Output Arguments

suite

Set of tests, specified as a matlab.unittest.TestSuite array.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Run a Single Test Method

Add the matlab.unittest.TestSuite class to the current import list.

```
import matlab.unittest.TestSuite;

cls = ?mypackage.MyTestClass;

% Create the suite using the method name
suite = TestSuite.fromMethod(cls, 'testMethod');
result = run(suite)

% Create the suite using the meta.method instance
metaMethod = findobj(cls.MethodList, 'Name', 'testMethod');
suite = TestSuite.fromMethod(cls, metaMethod);
result = run(suite)
```

See Also

`TestRunner` | `fromClass` | `fromPackage` |

Purpose

Create `TestSuite` array from all tests in package

Syntax

```
suite = matlab.unittest.TestSuite.fromPackage(package)
```

Description

`suite = matlab.unittest.TestSuite.fromPackage(package)` creates a TestSuite array from all of the `Test` methods of all concrete TestCase classes contained in `package` and returns that array in `suite`. The method is not recursive, returning only those tests directly in the package specified.

Tips

- The root folder(s) where `package` is defined must be on the MATLAB path when creating `suite` using this method as well as when `suite` is run.

Input Arguments

package

The name of the desired package to find tests, specified as a string.

Output Arguments

suite

Set of tests, specified as a matlab.unittest.TestSuite array.

Attributes

`Static`	`true`

To learn about attributes of methods, see Method Attributes in the MATLAB Object-Oriented Programming documentation.

Examples

Run All Tests in a Package

Add the matlab.unittest.TestSuite class to the current import list.

```
import matlab.unittest.TestSuite;

suite = TestSuite.fromPackage('mypackage.subpackage');
result = run(suite)
```

See Also

`TestRunner` | `fromMethod` | `fromClass` |

Purpose

Run `TestSuite` array using `TestRunner` object configured for text output

Syntax

```
result = run(suite)
```

Description

`result = run(suite)` runs the TestSuite object defined by `suite` using a TestRunner object configured for text output.

Tips

- This is a convenience method which is equivalent to using a TestRunner object created from the TestRunner.withTextOutput method to run `suite`.

Input Arguments

suite

Set of tests, specified as a matlab.unittest.TestSuite array.

Output Arguments

result

A matlab.unittest.TestResult object containing the result of the test run. `result` is the same size as `suite` and each element is the result of the corresponding element in `suite`.

Examples

Compare TestSuite.run with TestRunner.run

Add matlab.unittest classes to the current import list.

```
import matlab.unittest.TestRunner;
import matlab.unittest.TestSuite;
```

Create a test suite and a test runner.

```
suite = TestSuite.fromClass(?mypackage.MyTestClass);
runner = TestRunner.withTextOutput;
```

The following test results are equivalent.

```
result = runner.run(suite)
result = run(suite)
```

See Also

`matlab.unittest.TestRunner.run` | `matlab.unittest.TestCase.run` |

Purpose

Format text into TeX string

Syntax

```
TeXString = texlabel(f)
TeXString = texlabel(f,'literal')
```

Description

`TeXString = texlabel(f)` converts the MATLAB expression `f` into the TeX equivalent for use in text strings. `texlabel` converts Greek variable names (for example, lambda, delta, and so on) into a string that is displayed as Greek letters. The `TeXString` output is useful as an argument to annotation functions such as `title`, `xlabel`, and `text`.

If `TeXString` is too long to fit into a figure window, then the center of the expression is replaced with a tilde ellipsis (~~~).

`TeXString = texlabel(f,'literal')` interprets Greek variable names literally.

Input Arguments

f - Input MATLAB expression

string

Input MATLAB expression, specified as a string.

Example: `'theta (degrees)'`

Data Types

`char`

Examples

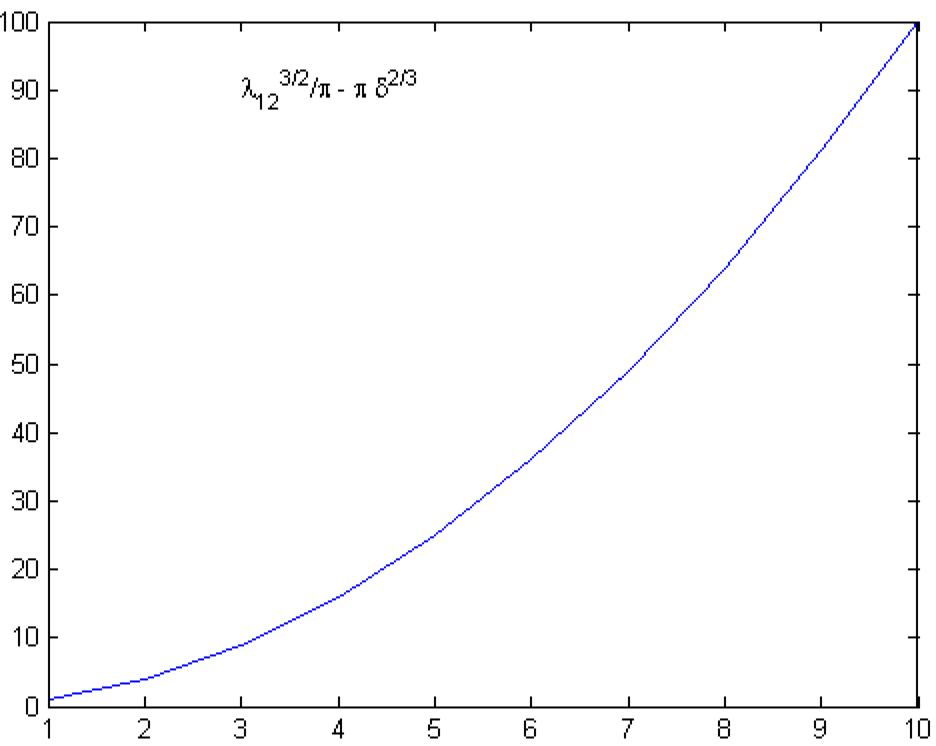
Insert TeX String in Figure

Create a figure and a TeX string for use in a text graphics object.

```
figure
plot((1:10).^2)
TeXString = texlabel('lambda12^(3/2)/pi - pi*delta^(2/3)');
```

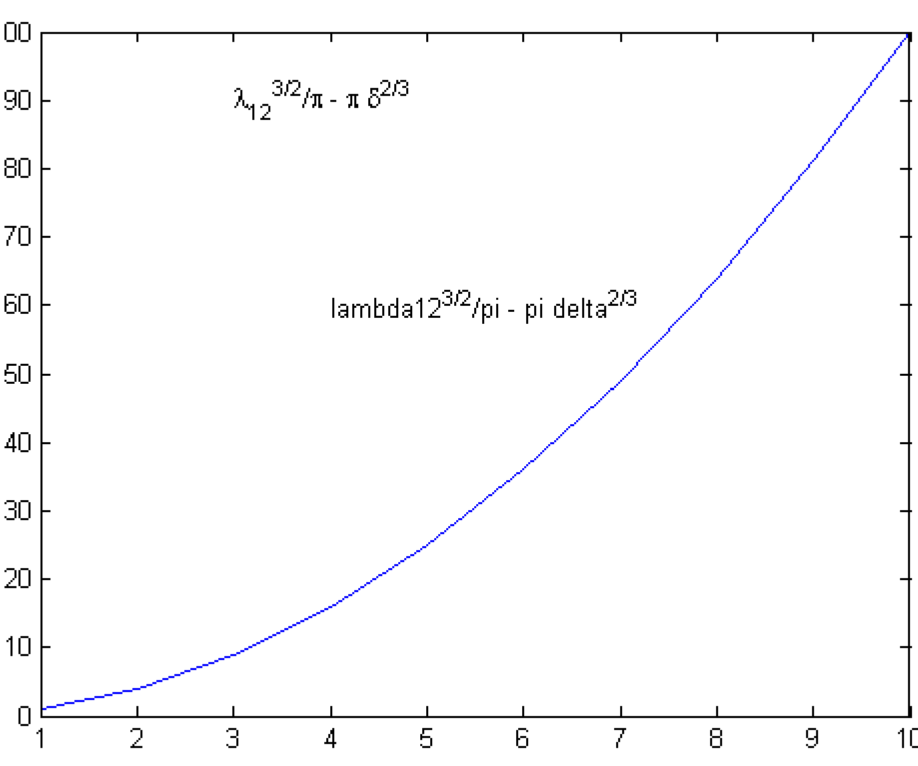
Add a text object containing the TeX string to the figure.

```
text(3,90,TeXString)
```



If you include the `'literal'` argument, `texlabel` interprets Greek variable names literally. Add a text object containing the literal string.

```
text(4,60,texlabel('lambda12^(3/2)/pi - pi*delta^(2/3)','literal'))
```



See Also

`text` | `title` | `xlabel` | `ylabel` | `zlabelString`

Purpose

Create text object in current axes

Syntax

```
text(x,y,'string')
text(x,y,z,'string')
text(x,y,z,'string','PropertyName',PropertyValue....)
text('PropertyName',PropertyValue....)
h = text(...)
```

Properties

For a list of properties, see `Text Properties`.

Description

`text` is the low-level function for creating text graphics objects. Use `text` to place character strings at specified locations.

`text(x,y,'string')` adds the string in quotes to the location specified by the point `(x,y)` `x` and `y` must be numbers of class `double`.

`text(x,y,z,'string')` adds the string in 3-D coordinates. `x`, `y` and `z` must be numbers of class `double`.

`text(x,y,z,'string','`*`PropertyName`*`',PropertyValue....)` adds the string in quotes to the location defined by the coordinates and uses the values for the specified text properties. For a description of the properties, see `Text Properties`.

`text('`*`PropertyName`*`',PropertyValue....)` omits the coordinates entirely and specifies all properties using property name/property value pairs.

`h = text(...)` returns a column vector of handles to text objects, one handle per object. All forms of the `text` function optionally return this output argument.

See the `String` property for a list of symbols, including Greek letters.

Tips

Position Text Within the Axes

The default text units are the units used to plot data in the graph. Specify the text location coordinates (the `x`, `y`, and `z` arguments) in the data units of the current graph (see “Examples” on page 1-5518. You can use other units to position the text by setting the text `Units`

property to `normalized` or one of the nonrelative units (`pixels`, `inches`, `centimeters`, `points`, or `characters`).

Note that the Axes Units property controls the positioning of the Axes within the figure and is not related to the axes data units used for graphing.

The `Extent`, `VerticalAlignment`, and `HorizontalAlignment` properties control the positioning of the character string with regard to the text location point.

If the coordinates are vectors, `text` writes the string at all locations defined by the list of points. If the character string is an array the same length as `x`, `y`, and `z`, `text` writes the corresponding row of the string array at each point specified.

Multiline Text

When specifying strings for multiple text objects, the string can be:

- A cell array of strings
- A padded string matrix

Each element of the specified string array creates a different text object.

When specifying the string for a single text object, cell arrays of strings and padded string matrices result in a text object with a multiline string, while vertical slash characters are not interpreted as separators and result in a single line string containing vertical slashes.

Behavior of the Text Function

`text` is a low-level function that accepts property name/property value pairs as input arguments. However, the convenience form:

```
text(x,y,z,'string')
```

is equivalent to:

```
text('Position',[x,y,z],'String','string')
```

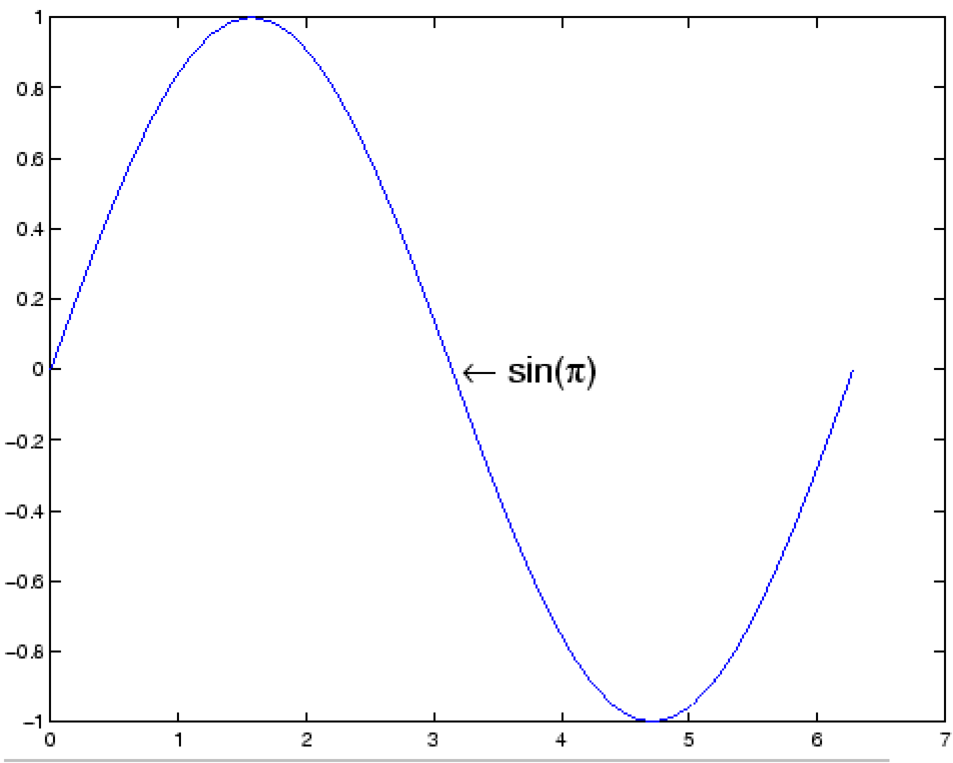
You can specify other properties only as property name/property value pairs. For a description of each property, see Text Properties. You can specify properties as property name/property value pairs, structure arrays, and cell arrays (see the set and get reference pages for examples of how to specify these data types).

text does not respect the setting of the figure or axes NextPlot property. This allows you to add text objects to an existing axes without setting hold to on.

Examples

Annotate the point (pi,0) with the string sin(π) on the following graph:

```
plot(0:pi/20:2*pi,sin(0:pi/20:2*pi))
text(pi,0,' \leftarrow sin(\pi)','FontSize',18)
```



Use embedded TeX sequences to display an equation:

```
axes
text(0.5,0.5,'\ite^{i\omega\tau} = cos(\omega\tau) + i sin(\omega\tau)')
```

MATLAB displays:

$$e^{i\omega\tau} = \cos(\omega\tau) + i\sin(\omega\tau)$$

Setting Default Properties

You can set default text properties on the axes, figure, and `root object` levels:

```
set(0,'DefaulttextProperty',PropertyValue...)
set(gcf,'DefaulttextProperty',PropertyValue...)
set(gca,'DefaulttextProperty',PropertyValue...)
```

Where *`Property`* is the name of the text property and `PropertyValue` is the value you are specifying. Use `set` and `get` to access text properties.

See Also

`annotation` | `gtext` | `int2str` | `num2str` | `strings` | `title` | `xlabel` | `ylabel` | `zlabel` | `Text Properties`

Purpose

Text properties

Creating Text Objects

Use `text` to create text objects.

Modifying Properties

You can set and query graphics object properties using the property editor or the `set` and `get` commands.

- Property Editor is an interactive tool that enables you to see and change object property values.
- The `set` and `get` commands enable you to set and query the values of properties.

To change the default values of properties, see Setting Default Property Values.

See Core Objects for general information about this type of object.

Text Property Descriptions

This section provides a description of properties. Curly braces `{ }` enclose default values.

`Annotation`

`hg.Annotation` object (read-only)

Handle of `Annotation` object. The `Annotation` property enables you to specify whether this text object is represented in a figure legend.

Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object.

Once you have obtained the `hg.LegendEntry` object, you can set its `IconDisplayStyle` property to control whether the text object is displayed in a figure legend:

IconDisplayStyle Value	Purpose
on	Represent this text object in a legend (default)
off	Do not include this text object in a legend
children	Same as on because text objects do not have children

Setting the IconDisplayStyle property

Set the IconDisplayStyle of a graphics object with handle hobj to off:

```
hAnnotation = get(hobj,'Annotation');
hLegendEntry = get(hAnnotation,'LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

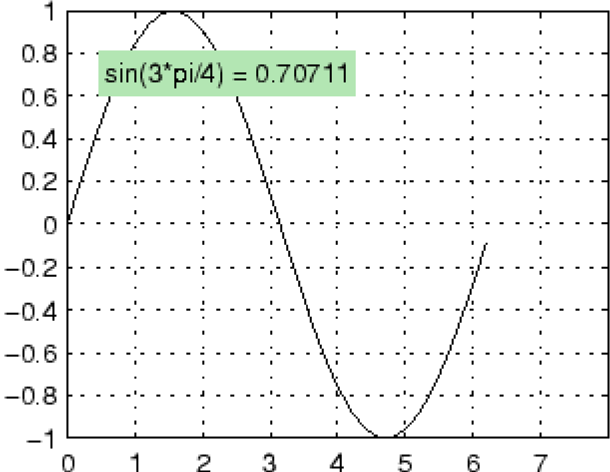
Using the IconDisplayStyle property

See "Controlling Legends" for more information and examples.

BackgroundColor
ColorSpec | {none}

Color of text extent rectangle. Defines a color for the rectangle that encloses the text Extent plus the text Margin. For example, the following code creates a text object that labels a plot and sets the background color to light green.

```
text(3*pi/4,sin(3*pi/4),...
 ['sin(3*pi/4) = ',num2str(sin(3*pi/4))],...
 'HorizontalAlignment','center',...
 'BackgroundColor',[.7 .9 .7]);
```



For additional features, see the following properties:

- `EdgeColor` — Color of the rectangle's edge (`none` by default).
- `LineStyle` — Style of the rectangle's edge line (first set `EdgeColor`)
- `LineWidth` — Width of the rectangle's edge line (first set `EdgeColor`)
- `Margin` — Increase the size of the rectangle by adding a margin to the existing text extent rectangle. This margin is added to the text extent rectangle to define the text background area that is enclosed by the `EdgeColor` rectangle. Note that the text extent does not change when you change the margin; only the rectangle displayed when you set the `EdgeColor` property and the area defined by the `BackgroundColor` change.

See also Drawing Text in a Box in the MATLAB Graphics documentation for an example using background color with contour labels.

`BeingDeleted`
`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Button press callback function. Executes whenever you press a mouse button while the pointer is over the text object.

See the figure's `SelectionType` property to determine if modifier keys were also pressed.

Set this property to a function handle that references the callback. The function must define at least two input arguments (handle of object associated with the button down event and an event structure, which is empty for this property).
The following example shows how to access the callback object's handle as well as the handle of the figure that contains the object from the callback function.

```
function button_down(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
   sel_typ = get(gcbf,'SelectionType')
   switch sel_typ
      case 'normal'
         disp('User clicked left-mouse button')
         set(src,'Selected','on')
      case 'extend'
         disp('User did a shift-click')
         set(src,'Selected','on')
      case 'alt'
         disp('User did a control-click')
         set(src,'Selected','on')
         set(src,'SelectionHighlight','off')
   end
end
```

Suppose `h` is the handle of a text object and the `button_down` function is on your MATLAB path. The following statement assigns the `button_down` function to the `ButtonDownFcn` property:

```
set(h,'ButtonDownFcn',@button_down)
```

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`Children`
: matrix (read-only)

The empty matrix; text objects have no children.

`Clipping`
: `on | {off}`

Clipping mode. When `Clipping` is `on`, MATLAB does not display any portion of the text that is outside the axes.

`Color`
: `ColorSpec`

Text color. A three-element RGB vector or one of the predefined names, specifying the text color. The default value is `[0 0 0]` (`black`). See the `ColorSpec` reference page for more information on specifying color.

`CreateFcn`
: function handle | cell array containing function handle and additional arguments | string (not recommended)

Callback function executed during object creation. Executes when MATLAB creates a text object. You must define this property as a default value for text or in a call to the `text` function that creates a new text object. For example, the statement:

```
set(0,'DefaultTextCreateFcn',@text_create)
```

defines a default value on the root level that sets the figure `Pointer` property to crosshairs whenever you create a text object. The callback function must be on your MATLAB path when you execute the above statement.

```
function text_create(src,evnt)
% src - the object that is the source of the event
% evnt - empty for this property
 set(gcbf,'Pointer','crosshair')
end
```

MATLAB executes this function after setting all text properties. Setting this property on an existing text object has no effect. The function must define at least two input arguments (handle of object created and an event structure, which is empty for this property).

The handle of the object whose `CreateFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DeleteFcn`

function handle | cell array containing function handle and additional arguments | string (not recommended)

Delete text callback function. Executes when you delete the text object (for example, when you issue a `delete` command or clear the axes `cla` or figure `clf`).

For example, the following function displays object property data before the object is deleted.

```
function delete_fcn(src,evnt)
% src - the object that is the source of the event
```

```
% evnt - empty for this property
   obj_tp = get(src,'Type');
   disp([obj_tp, ' object deleted'])
   disp('Its user data is:')
   disp(get(src,'UserData'))
end
```

MATLAB executes the function before deleting the object's properties so these values are available to the callback function. The function must define at least two input arguments (handle of object being deleted and an event structure, which is empty for this property).

The handle of the object whose `DeleteFcn` is being executed is passed by MATLAB as the first argument to the callback function and is also accessible through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DisplayName`
string

String used by legend. The `legend` function uses the `DisplayName` property to label the text object in the legend. The default is an empty string.

- If you specify string arguments with the `legend` function, MATLAB set `DisplayName` to the corresponding string and uses that string for the legend.
- If `DisplayName` is empty, `legend` creates a string of the form, `['data' n]`, where *n* is the number assigned to the object based on its location in the list of legend entries. However, `legend` does not set `DisplayName` to this string.
- If you edit the string directly in an existing legend, MATLAB set `DisplayName` to the edited string.

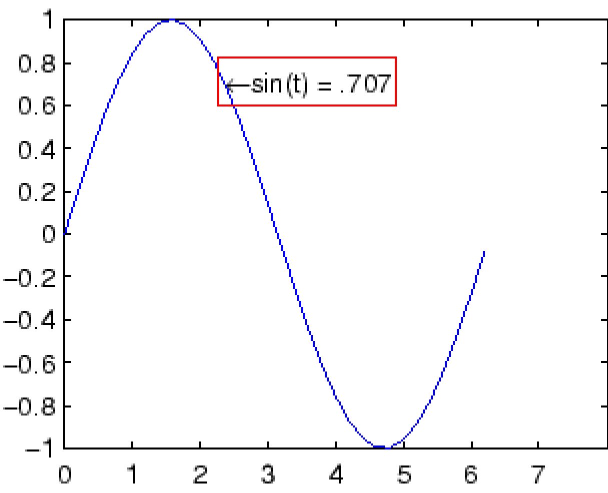
- If you specify a string for the `DisplayName` property and create the legend using the figure toolbar, then MATLAB uses the string defined by `DisplayName`.
- To add a legend programmatically that uses the `DisplayName` string, call `legend` with the `toggle` or `show` option.

See "Controlling Legends" for more information and examples.

`EdgeColor`
`ColorSpec | {none}`

Color of edge drawn around text extent rectangle plus margin. Specifies the color of a box drawn around the text `Extent` plus the text `Margin`. For example, the following code draws a red rectangle around text that labels a plot.

```
text(3*pi/4,sin(3*pi/4),...
'\leftarrowsin(t) = .707',...
'EdgeColor','red');
```



For additional features, see the following properties:

- `BackgroundColor` — Color of the rectangle's interior (`none` by default)
- `LineStyle` — Style of the rectangle's edge line (first set `EdgeColor`)
- `LineWidth` — Width of the rectangle's edge line (first set `EdgeColor`)
- `Margin` — Increases the size of the rectangle by adding a margin to the area defined by the text extent rectangle. This margin is added to the text extent rectangle to define the text background area that is enclosed by the `EdgeColor` rectangle. Note that the text extent does not change when you change the margin; only the rectangle displayed when you set the `EdgeColor` property and the area defined by the `BackgroundColor` change.

`Editing`
`on | {off}`

Enable or disable editing mode. When this property is `off` (the default), you cannot edit the text string interactively (i.e., you must change the `String` property to change the text). When you set this property to `on`, MATLAB enables editing and places an insert cursor wherever the mouse is within the text. To apply the new text string, do any one of the following:

- Press the **Esc** key.
- Click anywhere away from the text string.
- Reset the `Editing` property to `off.`

MATLAB updates the `String` property to contain the new text and resets the `Editing` property to `off`. You must reset the `Editing` property to `on` to resume editing.

`EraseMode`
`{normal} | none | xor | background`

Erase mode. Controls the technique MATLAB uses to draw and erase text objects. Alternative erase modes are useful for creating animated sequences where controlling the way individual objects are redrawn is necessary to improve performance and obtain the desired effect.

- `normal` — Redraw the affected region of the display, performing the three-dimensional analysis necessary to ensure that all objects are rendered correctly. This mode produces the most accurate picture, but is the slowest. The other modes are faster, but do not perform a complete redraw and are therefore less accurate.
- `none` — Do not erase the text when it is moved or destroyed. While the object is still visible on the screen after erasing with `EraseMode none`, you cannot print it because MATLAB stores no information about its former location.
- `xor` — Draw and erase the text by performing an exclusive OR (XOR) with each pixel index of the screen beneath it. When the text is erased, it does not damage the objects beneath it. However, when text is drawn in `xor` mode, its color depends on the color of the screen beneath it. It is correctly colored only when it is over axes background `Color`, or the figure background `Color` if the axes `Color` is `none`.
- `background` — Erase the text by drawing it in the axes background `Color`, or the figure background `Color` if the axes `Color` is `none`. This damages objects that are behind the erased text, but text is always properly colored.

Printing with Nonnormal Erase Modes

MATLAB always prints figures as if the `EraseMode` of all objects is `normal`. This means graphics objects created with `EraseMode` set to `none`, `xor`, or `background` can look different on screen than on paper. On screen, MATLAB mathematically combines layers of colors (for example, performing an XOR on a pixel color with that of the pixel behind it) and ignore three-dimensional sorting

to obtain greater rendering speed. However, these techniques are not applied to the printed output.

You can use the `getframe` command or other screen capture applications to create an image of a figure containing nonnormal mode objects.

`Extent`

position rectangle (read-only)

Position and size of text. A four-element vector that defines the size and position of the text string:

`[left,bottom,width,height]`

If the `Units` property is `data` (the default), `left` and `bottom` are the *x*- and *y*-coordinates of the lower left corner of the text `Extent`.

For all other values of `Units`, `left` and `bottom` are the distance from the lower left corner of the axes `Position` rectangle to the lower left corner of the text `Extent`. `width` and `height` are the dimensions of the `Extent` rectangle. All measurements are in units specified by the `Units` property.

`FontAngle`

`{normal} | italic | oblique`

Character slant. MATLAB uses this property to select a font from those available on your particular system. Generally, setting this property to `italic` or `oblique` selects a slanted font.

`FontName`

name (such as `Courier`) | `FixedWidth`

Font family. Specifies the name of the font to use for the text object. To display and print properly, this must be a font that your system supports. The default font is `Helvetica`.

Specifying a Fixed-Width Font

If you want text to use a fixed-width font that looks good in any locale, you should set FontName to the string FixedWidth:

```
set(text_handle,'FontName','FixedWidth')
```

This eliminates the need to hard-code the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding (such as in Japan where multibyte character sets are used). A properly written MATLAB application that needs to use a fixed-width font should set FontName to FixedWidth (note that this string is case sensitive) and rely on FixedWidthFontName to be set correctly in the end user's environment.

End users can adapt a MATLAB application to different locales or personal environments by setting the root FixedWidthFontName property to the appropriate value for that locale from startup.m.

Note that setting the root FixedWidthFontName property causes an immediate update of the display to use the new font.

FontSize
size in FontUnits

Font size. Specifies the font size to use for text in units determined by the FontUnits property. The default is 10 points. 1 point = $^1/_{72}$ inch.

FontWeight
{normal} | bold | light | demi

Weight of text characters. MATLAB uses this property to select a font from those available on your particular system. Generally, setting this property to bold or demi causes MATLAB to use a bold font.

`FontUnits`
`{points} | normalized | inches | centimeters | pixels`

Font size units. MATLAB uses this property to determine the units used by the `FontSize` property.

- `normalized` — Interpret `FontSize` as a fraction of the height of the parent axes. When you resize the axes, MATLAB modifies the screen `FontSize` accordingly.
- `pixels`, `inches`, `centimeters`, and `points` — Absolute units. 1 point = $^1/_{72}$ inch.

Note that if you are setting both the `FontSize` and the `FontUnits` in one function call, you must set the `FontUnits` property first so that MATLAB can correctly interpret the specified `FontSize`.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. Determines when an object's handle is visible in its parent's list of children. `HandleVisibility` is useful for preventing command-line users from accidentally drawing into or deleting a figure that contains only user interface devices (such as a dialog box).

- `on` — Handles are always visible.
- `callback` — Handles are visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have access to object handles.
- `off` — Handles are invisible at all times. Use this option when a callback invokes a function that could damage the GUI (such as evaluating a user-typed string). This option temporarily hides its own handles during the execution of that function.

When a handle is not visible in its parent's list of children, it cannot be returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`.

When a handle's visibility is restricted using `callback` or `off`,

- The object's handle does not appear in its parent's `Children` property.
- Figures do not appear in the root's `CurrentFigure` property.
- Objects do not appear in the root's `CallbackObject` property or in the figure's `CurrentObject` property.
- Axes do not appear in their parent's `CurrentAxes` property.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible regardless of their `HandleVisibility` settings (this does not affect the values of the `HandleVisibility` properties).

Handles that are hidden are still valid. If you know an object's handle, you can set and get its properties, and pass it to any function that operates on handles.

`HitTest`
`{on} | off`

Selectable by mouse click. Determines if the text can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the text. If `HitTest` is `off`, clicking the text selects the object below it (which is usually the axes containing it).

For example, suppose you define the button down function of an image (see the `ButtonDownFcn` property) to display text at the location you click with the mouse.

First define the callback routine.

```
function bd_function
pt = get(gca,'CurrentPoint');
text(pt(1,1),pt(1,2),pt(1,3),...
 '{\fontsize{20}\oplus} The spot to label',...
 'HitTest','off')
```

Now display an image, setting its `ButtonDownFcn` property to the callback routine.

```
load earth
image(X,'ButtonDownFcn','bd_function'); colormap(map)
```

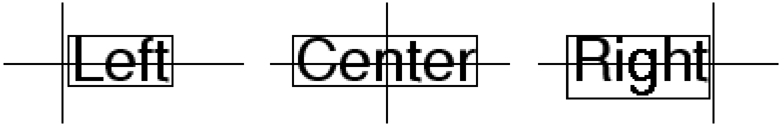
When you click the image, MATLAB displays the text string at that location. With `HitTest` set to `off`, existing text cannot intercept any subsequent button down events that occur over the text. This enables the image's button down function to execute.

`HorizontalAlignment`

`{left} | center | right`

Horizontal alignment of text. Specifies the horizontal justification of the text string. It determines where MATLAB places the string with regard to the point specified by the `Position` property. The following picture illustrates the alignment options.

`HorizontalAlignment` **viewed with the** `VerticalAlignment` **set to middle (the default).**



See the `Extent` property for related information.

See "Text Alignment" for more information and examples.

`Interpreter`

`latex | {tex} | none`

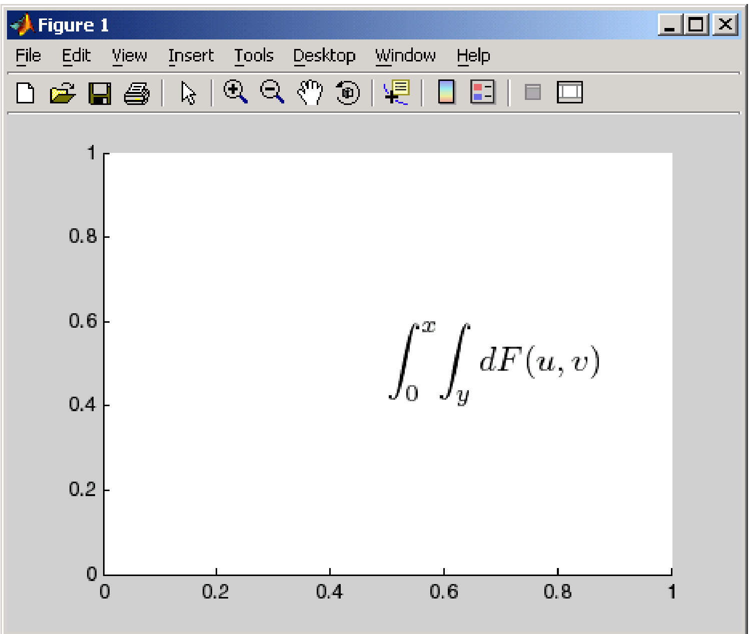
Interpret TeXinstructions. Controls whether MATLAB interprets certain characters in the `String` property as TeX instructions (default) or displays all characters literally.

- `latex` — Supports a basic subset of the LaTeX markup language.
- `tex` — Supports a subset of plain TeX markup language. See the `String` property for a list of supported TeX instructions.
- `none` — Displays literal characters.

LaTeX Interpreter

To enable the LaTeX interpreter for text objects, set the `Interpreter` property to `latex`. For example, the following statement displays an equation in a figure at the point `[.5 .5]`, and enlarges the font to 16 points.

```
text('Interpreter','latex',...
 'String','$$\int_0^x\!\int_y dF(u,v)$$',...
 'Position',[.5 .5],...
 'FontSize',16)
```



Note The maximum size of the string that you can use with the LaTeX interpreter is 1200 characters. For multiline strings, reduce this amount by about 10 characters per line.

Information About Using TeX

The following references may be useful to people who are not familiar with TeX.

- Donald E. Knuth, *The TEXbook*, Addison Wesley, 1986.
- The TeX Users Group home page: http://www.tug.org

```
Interruptible
    off | {on}
```

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For Graphics objects, the `Interruptible` property affects only the callbacks for the`ButtonDownFcn` property. A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both the callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

`BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting `Interruptible` property to `on` (default), allows a callback from other graphics objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see "Control Callback Execution and Interruption".

`LineStyle`

`{-} | -- | : | -. | none`

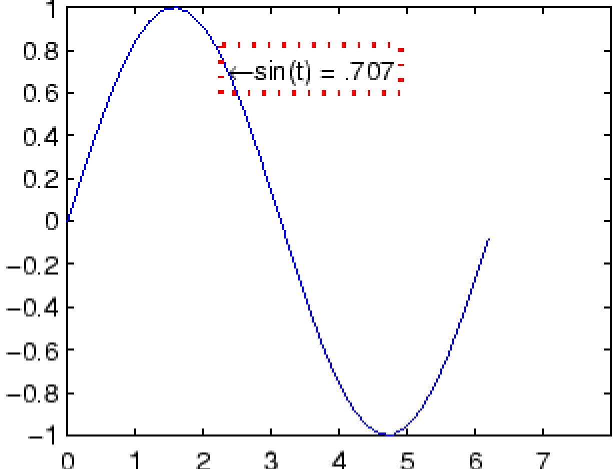
Edge line type. Specifies the line style used to draw the edges of the text `Extent`.

Line Style Specifiers Table

Specifier	Line Style
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line
'none'	No line

For example, the following code draws a red rectangle with a dotted line style around text that labels a plot.

```
text(3*pi/4,sin(3*pi/4),...
     '\leftarrowsin(t) = .707',...
     'EdgeColor','red',...
     'LineWidth',2,...
     'LineStyle',':');
```



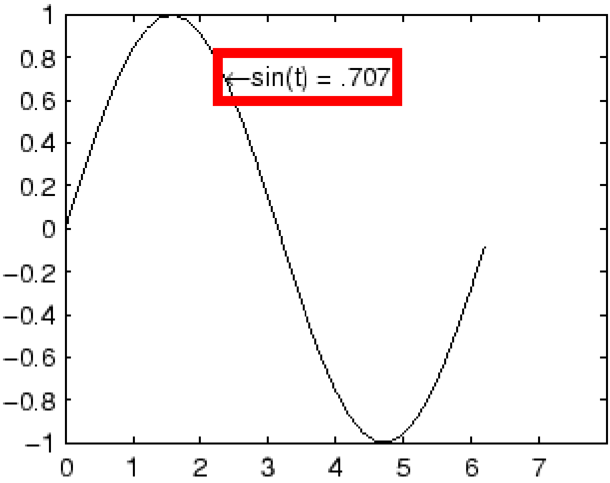
For additional features, see the following properties:

- `BackgroundColor` — Color of the rectangle's interior (`none` by default)
- `EdgeColor` — Color of the rectangle's edge (`none` by default)
- `LineWidth` — Width of the rectangle's edge line (first set `EdgeColor`)
- `Margin` — Increases the size of the rectangle by adding a margin to the existing text extent rectangle. This margin is added to the text extent rectangle to define the text background area that is enclosed by the `EdgeColor` rectangle. Note that the text extent does not change when you change the margin; only the rectangle displayed when you set the `EdgeColor` property and the area defined by the `BackgroundColor` change.

`LineWidth`
scalar (points)

Width of line used to draw text extent rectangle. When you set the text `EdgeColor` property to a color (the default is `none`), MATLAB displays a rectangle around the text `Extent`. Use the `LineWidth` property to specify the width of the rectangle edge. For example, the following code draws a red rectangle around text that labels a plot and specifies a line width of 3 points:

```
text(3*pi/4,sin(3*pi/4),...
'\leftarrowsin(t) = .707',...
'EdgeColor','red',...
'LineWidth',3);
```



For additional features, see the following properties:

- `BackgroundColor` — Color of the rectangle's interior (`none` by default)
- `EdgeColor` — Color of the rectangle's edge (`none` by default)
- `LineStyle` — Style of the rectangle's edge line (first set `EdgeColor`)

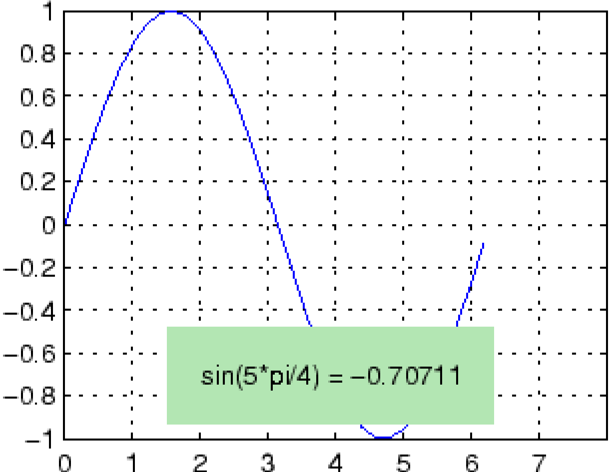
- `Margin` — Increases the size of the rectangle by adding a margin to the existing text extent rectangle. This margin is added to the text extent rectangle to define the text background area that is enclosed by the `EdgeColor` rectangle. Note that the text extent does not change when you change the margin; only the rectangle displayed when you set the `EdgeColor` property and the area defined by the `BackgroundColor` change.

`Margin`

scalar (pixels)

Distance between the text extent and the rectangle edge. When you specify a color for the `BackgroundColor` or `EdgeColor` text properties, MATLAB draws a rectangle around the area defined by the text `Extent` plus the value specified by the `Margin`. For example, the following code displays a light green rectangle with a 10-pixel margin.

```
text(5*pi/4,sin(5*pi/4),...
 ['sin(5*pi/4) = ',num2str(sin(5*pi/4))],...
 'HorizontalAlignment','center',...
 'BackgroundColor',[.7 .9 .7],...
 'Margin',10);
```



For additional features, see the following properties:

- `BackgroundColor` — Color of the rectangle's interior (`none` by default)
- `EdgeColor` — Color of the rectangle's edge (`none` by default)
- `LineStyle` — Style of the rectangle's edge line (first set `EdgeColor`)
- `LineWidth` — Width of the rectangle's edge line (first set `EdgeColor`)

See how margin affects text extent properties

This example enables you to change the values of the `Margin` property and observe the effects on the `BackgroundColor` area and the `EdgeColor` rectangle.

Click to view in editor — This link opens the MATLAB editor with the following example.

Click to run example — Use your scroll wheel to vary the `Margin`.

`Parent`
handle of axes, hggroup, or hgtransform

Parent of text object. Handle of the text object's parent. The parent of a text object is the axes, hggroup, or hgtransform object that contains it.

See "Objects That Can Contain Other Objects" for more information on parenting graphics objects.

`Position`
`[x,y,[z]]`

Location of text. A two- or three-element vector, `[x y [z]]`, that specifies the location of the text in three dimensions. If you omit the `z` value, it defaults to 0. All measurements are in units specified by the `Units` property. Initial value is `[0 0 0]`.

`Rotation`
scalar (default = 0)

Text orientation. Determines the orientation of the text string. Specify values of rotation in degrees (positive angles cause counterclockwise rotation).

`Selected`
`on | {off}`

Is object selected? When this property is `on`, MATLAB displays selection handles if the `SelectionHighlight` property is also set to `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`
`{on} | off`

Objects are highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by

drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`String`
string

Text string. Specify this property as a quoted string for single-line strings, or as a cell array of strings, or a padded string matrix for multiline strings. MATLAB displays this string at the specified location. Vertical slash characters are not interpreted as line breaks in text strings, and are drawn as part of the text string. See Mathematical Symbols, Greek Letters, and TeX Characters for an example.

Note The words `default`, `factory`, and `remove` are reserved words that will not appear in a figure when quoted as a normal string. In order to display any of these words individually, type `'\reserved_word'` instead of `'reserved_word'`.

When the text `Interpreter` property is `tex` (the default), you can use a subset of TeX commands embedded in the string to produce special characters such as Greek letters and mathematical symbols. The following table lists these characters and the character sequences used to define them.

TeX Character Sequence Table

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\alpha	α	\upsilon	υ	\sim	~
\angle	∠	\phi	Φ	\leq	≤
\ast	*	\chi	χ	\infty	∞
\beta	β	\psi	ψ	\clubsuit	♣

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\gamma	γ	\omega	ω	\diamondsuit	♦
\delta	δ	\Gamma	Γ	\heartsuit	♥
\epsilon	ε	\Delta	Δ	\spadesuit	♠
\zeta	ζ	\Theta	Θ	\leftrightarrow	↔
\eta	η	\Lambda	Λ	\leftarrow	←
\theta	Θ	\Xi	Ξ	\Leftarrow	⇐
\vartheta		\Pi	Π	\uparrow	↑
\iota	ι	\Sigma	Σ	\rightarrow	→
\kappa	κ	\Upsilon		\Rightarrow	⇒
\lambda	λ	\Phi	Φ	\downarrow	↓
\mu	µ	\Psi	Ψ	\circ	º
\nu	ν	\Omega	Ω	\pm	±
\xi	ξ	\forall	∀	\geq	≥
\pi	π	\exists	∃	\propto	∝
\rho	ρ	\ni	э	\partial	∂
\sigma	σ	\cong	≅	\bullet	•
\varsigma	ς	\approx	≈	\div	÷
\tau	τ	\Re	ℜ	\neq	≠
\equiv	≡	\oplus	⊕	\aleph	
\Im	ℑ	\cup	∪	\wp	℘
\otimes	⊗	\subseteq	⊆	\oslash	∅

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\cap	∩	\in		\supseteq	⊇
\supset	⊃	\lceil	⌈	\subset	⊂
\int	∫	\cdot	·	\o	ο
\rfloor	⌋	\neg	¬	\nabla	∇
\lfloor	⌊	\times	x	\ldots	...
\perp	⊥	\surd	√	\prime	′
\wedge	∧	\varpi	ϖ	\0	∅
\rceil	⌉	\rangle	〉	\mid	|
\vee	∨			\copyright	©
\langle	〈				

You can also specify stream modifiers that control font type and color. The first four modifiers are mutually exclusive. However, you can use `\fontname` in combination with one of the other modifiers:

- `\bf` — Bold font
- `\it` — Italic font
- `\sl` — Oblique font (rarely available)
- `\rm` — Normal font
- `\fontname{`*`fontname`*`}` — Specify the name of the font family to use.
- `\fontsize{`*`fontsize`*`}` — Specify the font size in `FontUnits`.

- `\color(`*colorSpec*`)` — Specify color for succeeding characters

  Stream modifiers remain in effect until the end of the string or only within the context defined by braces `{ }`.

Specifying Text Color in TeX Strings

Use the `\color` modifier to change the color of characters following it from the previous color (which is `black` by default). Syntax is:

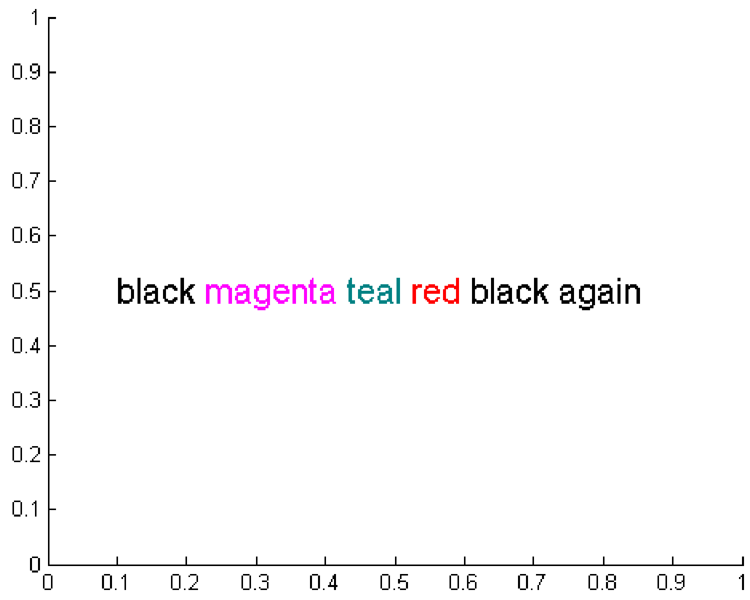
- `\color{colorname}` — Use for the eight basic named colors (`red`, `green`, `yellow`, `magenta`, `blue`, `black`, `white`), and plus the four Simulink colors (`gray`, `darkGreen`, `orange`, and `lightBlue`).

  Note that short names (one-letter abbreviations) for colors are not supported by the `\color` modifier.

- `\color[rgb]{r g b}` — Use to specify an RGB triplet with values between `0` and `1` as a cell array

For example:

```
text(.1,.5,['\fontsize{16}black {\color{magenta}magenta '...
'\color[rgb]{0 .5 .5}teal \color{red}red} black again'])
```



Specifying Subscript and Superscript Characters

The subscript character "_" and the superscript character "^" modify the character or substring defined in braces immediately following.

To print the special characters used to define the TeX strings when `Interpreter` is `tex`, prefix them with the backslash "\" character: \\, \{, \} _, \^.

See the "Examples" on page 1-5518 in the `text` reference page for more information.

When `Interpreter` is `none`, no characters in the `String` are interpreted, and all are displayed when the text is drawn.

When `Interpreter` is `latex`, MATLAB provides a complete LaTeX interpreter for text objects. See the `Interpreter` property for more information.

`Tag`

string

User-specified object label. Provides a means to identify graphics objects with a user-specified label. The default is an empty string.

Use the `Tag` property and the `findobj` function to manipulate specific objects within a plotting hierarchy.

`Type`

string (read-only)

Class of graphics object. String that identifies the class of the graphics object. Use this property to find all objects of a given type within a plotting hierarchy. For text objects, `Type` is always '`text`'.

`UIContextMenu`

handle of uicontextmenu object

Associate a context menu with the text. The handle of a uicontextmenu object created in the same figure as the text. Use the `uicontextmenu` function to create the context menu. MATLAB displays the context menu whenever you right-click over the text.

`Units`

`pixels | normalized | inches |`
`| characters | centimeters | points | {data}`

Units of measurement. Specifies the units MATLAB uses to interpret the `Extent` and `Position` properties. All units are measured from the lower left corner of the axes plot box.

- `normalized` — Units map the lower left corner of the rectangle defined by the axes to (0,0) and the upper right corner to (1.0,1.0).
- `pixels`, `inches`, `centimeters`, and `points` — Absolute units. 1 point = $^1/_{72}$ inch.

- `characters` — Based on the size of characters in the default system font. The width of one `characters` unit is the width of the letter `x`, and the height of one `characters` unit is the distance between the baselines of two lines of text.
- `data` — Data units of the parent axes as determined by the data graphed (not the axes `Units` property, which controls the positioning of the axes within the figure window).

If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is set to the default value.

`UserData`
matrix

User-specified data. Data you want to associate with the text object. The default value is an empty array. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`VerticalAlignment`
`top | cap | {middle} | baseline | bottom`

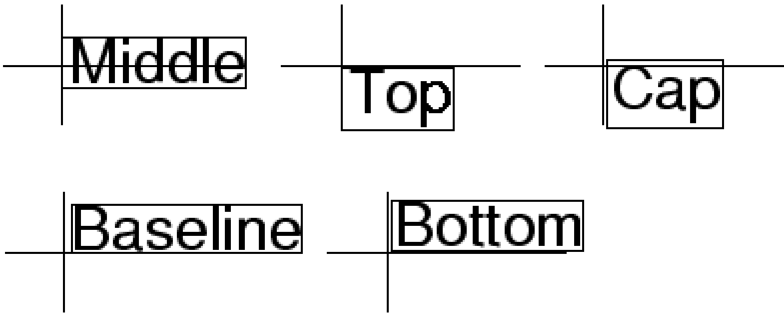
Vertical alignment of text. Specifies the vertical justification of the text string. It determines where MATLAB places the string vertically with regard to the points specified by the `Position` property.

- `top` — Place the top of the string' s `Extent` rectangle at the specified *y*-position.
- `cap` — Place the string so that the top of a capital letter is at the specified *y*-position.
- `middle` — Place the middle of the string at the specified *y*-position.
- `baseline` — Place font baseline at the specified *y*-position.

- `bottom` — Place the bottom of the string's `Extent` rectangle at the specified *y*-position.

The following picture illustrates the alignment options.

Text `VerticalAlignment` property viewed with the `HorizontalAlignment` property set to `left` (the default).



See "Text Alignment" for more information and examples.

`Visible`

`{on} | off`

Text visibility. By default, all text is visible. When set to `off`, the text is not visible, but still exists, and you can query and set its properties.

See Also

`text`

textread

Purpose

Read data from text file; write to multiple outputs

Note `textread` is not recommended. Use `textscan` instead.

Syntax

```
[A,B,C,...] = textread(filename,format)
[A,B,C,...] = textread(filename,format,N)
[...] = textread(...,param,value,...)
```

Description

`[A,B,C,...]  = textread(filename,format)` reads data from the file `filename` into the variables `A,B,C,` and so on, using the specified `format`, until the entire file is read. The `filename` and `format` inputs are strings, each enclosed in single quotes. `textread` is useful for reading text files with a known format. `textread` handles both fixed and free format files.

Note When reading large text files, reading from a specific point in a file, or reading file data into a cell array rather than multiple outputs, you might prefer to use the `textscan` function.

`textread` matches and converts groups of characters from the input. Each input field is defined as a string of non-white-space characters that extends to the next white-space or delimiter character, or to the maximum field width. Repeated delimiter characters are significant, while repeated white-space characters are treated as one.

The `format` string determines the number and types of return arguments. The number of return arguments is the number of items in the `format` string. The `format` string supports a subset of the conversion specifiers and conventions of the C language `fscanf` routine. Values for the `format` string are listed in the table below. White-space characters in the `format` string are ignored.

format	Action	Output
Literals (ordinary characters)	Ignore the matching characters. For example, in a file that has `Dept` followed by a number (for department number), to skip the `Dept` and read only the number, use `'Dept'` in the `format` string.	None
`%d`	Read a signed integer value.	Double array
`%u`	Read an integer value.	Double array
`%f`	Read a floating-point value.	Double array
`%s`	Read a white-space or delimiter-separated string.	Cell array of strings
`%q`	Read a double quoted string, ignoring the quotes.	Cell array of strings
`%c`	Read characters, including white space.	Character array
`%[...]`	Read the longest string containing characters specified in the brackets.	Cell array of strings
`%[^...]`	Read the longest nonempty string containing characters that are not specified in the brackets.	Cell array of strings
`%*...` `instead of %`	Ignore the matching characters specified by `*`.	No output
`%w...` `instead of %`	Read field width specified by `w`. The `%f` format supports `%w.pf`, where `w` is the field width and `p` is the precision.	

`[A,B,C,...] = textread(filename,format,N)` reads the data, reusing the `format` string `N` times, where `N` is an integer greater than zero. If `N` is smaller than zero, `textread` reads the entire file.

`[...] = textread(...,param,value,...)` customizes `textread` using `param/value` pairs, as listed in the table below.

param	value	Action
bufsize	Positive integer	Specifies the maximum string length, in bytes. Default is 4095.
commentstyle	matlab	Ignores characters after %.
commentstyle	shell	Ignores characters after #.
commentstyle	c	Ignores characters between /* and */.
commentstyle	c++	Ignores characters after //.
delimiter	One or more characters	Act as delimiters between elements. Default is none.
emptyvalue	Scalar double	Value given to empty cells when reading delimited files. Default is 0.
endofline	Single character or '\r\n'	Character that denotes the end of a line. Default is determined from file
expchars	Exponent characters	Default is eEdD.
headerlines	Positive integer	Ignores the specified number of lines at the beginning of the file.
whitespace	Any from the list below: ' ' Space \b Backspace \n Newline \r Carriage return \t Horizontal tab	Treats vector of characters as white space. Default is ' \b\t'.

Note When `textread` reads a consecutive series of `whitespace` values, it treats them as one white space. When it reads a consecutive series of `delimiter` values, it treats each as a separate delimiter.

Tips

If you want to preserve leading and trailing spaces in a string, use the `whitespace` parameter as shown here:

```
textread('myfile.txt', '%s', 'whitespace', '')
ans =
    '   An  example      of preserving    spaces      '
```

Examples

Example 1 — Read All Fields in Free Format File Using %

The first line of `mydata.dat` is

```
Sally    Level1 12.34 45 Yes
```

Read the first line of the file as a free format file using the `%` format.

```
[names, types, x, y, answer] = textread('mydata.dat', ...
'%s %s %f %d %s', 1)
```

returns

```
names =
    'Sally'
types =
    'Level1'
x =
  12.34000000000000
y =
    45
answer =
    'Yes'
```

Example 2 — Read as Fixed Format File, Ignoring the Floating Point Value

The first line of `mydata.dat` is

```
Sally    Level1 12.34 45 Yes
```

Read the first line of the file as a fixed format file, ignoring the floating-point value.

```
[names, types, y, answer] = textread('mydata.dat', ...
'%9c %6s %*f %2d %3s', 1)
```

returns

```
names =
Sally
types =
    'Level1'
y =
    45
answer =
    'Yes'
```

`%*f` in the `format` string causes `textread` to ignore the floating point value, in this case, `12.34`.

Example 3 — Read Using Literal to Ignore Matching Characters

The first line of `mydata.dat` is

```
Sally    Type1 12.34 45 Yes
```

Read the first line of the file, ignoring the characters `Type` in the second field.

```
[names, typenum, x, y, answer] = textread('mydata.dat', ...
'%s Type%d %f %d %s', 1)
```

returns

```
names =
    'Sally'
typenum =
    1
x =
  12.34000000000000
y =
    45
```

```
answer =
    'Yes'
```

`Type%d` in the `format` string causes the characters `Type` in the second field to be ignored, while the rest of the second field is read as a signed integer, in this case, `1`.

Example 4 — Specify Value to Fill Empty Cells

For files with empty cells, use the `emptyvalue` parameter. Suppose the file `data.csv` contains:

```
1,2,3,4,,6
7,8,9,,11,12
```

Read the file using `NaN` to fill any empty cells:

```
data = textread('data.csv', '', 'delimiter', ',', ...
                'emptyvalue', NaN);
```

Example 5 — Read File into a Cell Array of Strings

Read the file `fft.m` into cell array of strings.

```
file = textread('fft.m', '%s', 'delimiter', '\n', ...
                'whitespace', '');
```

See Also

`textscan` | `dlmread` | `fscanf`

Purpose

Read formatted data from text file or string

Syntax

```
C = textscan(fileID,formatSpec)
C = textscan(fileID,formatSpec,N)

C = textscan(str,formatSpec)
C = textscan(str,formatSpec,N)

C = textscan(___,Name,Value)

[C,position] = textscan(___)
```

Description

`C = textscan(fileID,formatSpec)` reads data from an open text file into cell array, `C`. The text file is indicated by the file identifier, `fileID`. Use `fopen` to open the file and obtain the `fileID` value. When you finish reading from a file, close the file by calling `fclose(fileID)`.

`textscan` attempts to match the data in the file to `formatSpec`, which is a string of conversion specifiers.

`C = textscan(fileID,formatSpec,N)` reads file data, using the `formatSpec` `N` times, where `N` is a positive integer. To read additional data from the file after `N` cycles, call `textscan` again using the original `fileID`. If you resume a text scan of a file by calling `textscan` with the same file identifier (`fileID`), then `textscan` automatically resumes reading at the point where it terminated the last read.

`C = textscan(str,formatSpec)` reads data from a string, `str` into cell array `C`. For strings, repeated calls to `textscan` restart the scan from the beginning each time. To restart a scan from the last position, request a `position` output.

`textscan` attempts to match the data in the string, `str`, to `formatSpec`, which is a string of conversion specifiers.

`C = textscan(str,formatSpec,N)` reads string data, using the `formatSpec` `N` times, where `N` is a positive integer.

`C = textscan(___,Name,Value)` specifies options using one or more `Name,Value` pair arguments, in addition to any of the input arguments in the previous syntaxes.

`[C,position] = textscan(___)` returns the file or string position at the end of the scan as the second output argument, using any of the input arguments in the previous syntaxes. For a file, this is the value that `ftell(fileID)` would return after calling `textscan`. For a string, `position` indicates how many characters `textscan` read.

Input Arguments

fileID - File identifier

numeric scalar

File identifier of an open text file, specified as a number. Before reading a file with `textscan`, you must use `fopen` to open the file and obtain the `fileID`.

Data Types

`double`

formatSpec - Format of the data fields

string

Format of the data fields, specified as a string of one or more conversion specifiers. When `textscan` reads a file or string, it attempts to match the data to the `formatSpec` string. If `textscan` fails to match a data field, it stops reading and returns all fields read before the failure.

The number of conversion specifiers determines the number of cells in output array, `C`.

Numeric Fields

This table lists available conversion specifiers for numeric inputs.

Numeric Input Type	Conversion Specifier	Output Class
Integer, signed	%d	int32
	%d8	int8
	%d16	int16
	%d32	int32
	%d64	int64
Integer, unsigned	%u	uint32
	%u8	uint8
	%u16	uint16
	%u32	uint32
	%u64	uint64
Floating-point number	%f	double
	%f32	single
	%f64	double
	%n	double

Character Fields

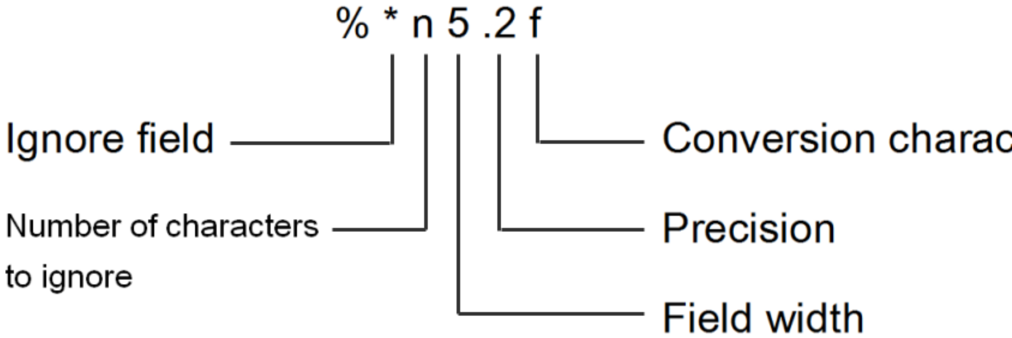
This table lists available conversion specifiers for character inputs.

Character Strings	Conversion Specifier	Details
Characters	%s	String
	%q	String, where double quotation marks indicate text to keep together
	%c	Any single character, including a delimiter

Character Strings	Conversion Specifier	Details
Pattern-matching strings	%[...]	Read only the characters inside the brackets up to the first nonmatching character. To include] in the set, specify it first: %[]...]. Example: %[mus] reads 'summer ' as 'summ'.
	%[^...]	Exclude characters inside the brackets, reading until the first matching character. To exclude], specify it first: %[^]...]. Example: %[^xrg] reads 'summer ' as 'summe'.

Optional Operators

Conversion specifiers in formatSpec can include optional operators, which appear in the following order (includes spaces for clarity):



Optional operators include:

- Fields and Characters to Ignore

textscan reads all characters in your file in sequence, unless you tell it to ignore a particular field or a portion of a field.

Use the following operators to skip or read portions of fields.

Operator	Action Taken
%*	Skip the field. textscan does not create an output cell for any field that it skips. Example: '%s %*s %s %s %*s %*s %s' (spaces are optional) converts the string 'Blackbird singing in the dead of night' to four output cells with the strings 'Blackbird' 'in' 'the' 'night'
%*n	Ignore n characters of the field, where n is an integer less than or equal to the number of characters in the field. Example: %*4s ignores 4 characters, so '%*4s %s' reads 'summer' as 'er'.

- Field Width

  textscan reads the number of characters or digits specified by the field width or precision, or up to the first delimiter, whichever comes first. A decimal point is counted as a digit. Specify the field width by inserting a number after the percent character (%) in the conversion specifier.

  Example: %5f reads '123.456' as 123.4.

Note When the field width operator is used with single characters (%c), textscan also reads delimiter characters.
Example: %7c reads 7 characters, including white-space, so'Day and night' reads as 'Day and'.

- Precision

  For floating-point numbers (`%n`, `%f`, `%f32`, `%f64`), you can specify the number of decimal digits to read.

  Example: `%7.2f` reads `'123.456'` as `123.45`.

- Literal Text to Ignore

  `textscan` ignores specified text appended to the `formatSpec` string.

  Example: `Level%u8` reads `'Level1'` as `1`.

  Example: `%u8Step` reads `'2Step'` as `2`.

N - Number of times to apply `formatSpec`

integer

Number of times to apply `formatSpec`, specified as an integer.

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

str - Input string

string

Input string to read.

Name-Value Pair Arguments

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

Names are not case sensitive.

Example: `C = textscan(fileID,formatSpec,'HeaderLines',3,'Delimiter',',')` skips the first three lines of the data, and then reads the remaining data, treating commas as a delimiter.

'CollectOutput' - Logical indicator determining data concatenation
`false` (default) | `true`

Logical indicator determining data concatenation, specified as the comma-separated pair consisting of `'CollectOutput'` and either `true` or `false`. If `true`, then `textscan` concatenates consecutive output cells of the same fundamental MATLAB class into a single array.

'CommentStyle' - Symbols designating text to ignore
string | cell array of strings

Symbols designating text to ignore, specified as the comma-separated pair consisting of `'CommentStyle'` and a string or cell array of strings.

For example, specify a string such as `'%'` to ignore characters following the string on the same line. Specify a cell array of two strings, such as `{'/*', '*/'}`, to ignore characters between the strings.

`textscan` checks for comments only at the start of each field, not within a field.

Example: `'CommentStyle',{'/*', '*/'}`

'Delimiter' - Field delimiter characters
`{' ','\b','\t'}` (default) | string | cell array of strings

Field delimiter characters, specified as the comma-separated pair consisting of `'Delimiter'` and a string or a cell array of strings. Specify multiple delimiters in a cell array of strings.

Example: `'delimiter',{';','*'}`

`textscan` interprets repeated delimiter characters as separate delimiters, and returns an empty value to the output cell.

Within each row of data, the default field delimiter is white space. White space can be any combination of space (`' '`), backspace (`'\b'`), or tab (`'\t'`) characters. If you do not specify a delimiter, `textscan` interprets repeated white-space characters as a single delimiter.

When you specify one of the following escape sequences as a delimiter, `textscan` converts that sequence to the corresponding control character:

`\b`	Backspace
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\\`	Backslash (\)

'EmptyValue' - Returned value for empty numeric fields

`NaN` (default) | scalar

Returned value for empty numeric fields in delimited text files, specified as the comma-separated pair consisting of `'EmptyValue'` and a scalar.

'EndOfLine' - End-of-line characters

string

End-of-line characters, specified as the comma-separated pair consisting of `'EndOfLine'` and a string. The default end-of-line sequence depends on the format of your file and can include a newline character (`'\n'`), a carriage return (`'\r'`), or a combination of the two (`'\r\n'`).

If there are missing values and an end-of-line sequence at the end of the last line in a file, then `textscan` returns empty values for those fields. This ensures that individual cells in output cell array, `C`, are the same size.

'ExpChars' - Exponent characters

`'eEdD'` (default) | string

Exponent characters, specified as the comma-separated pair consisting of `'ExpChars'` and a string. The default exponent characters are `e`, `E`, `d`, and `D`.

'HeaderLines' - Number of header lines

`0` (default) | positive integer

Number of header lines, specified as the comma-separated pair consisting of 'HeaderLines' and a positive integer. textscan skips the header lines, including the remainder of the current line.

'MultipleDelimsAsOne' - Multiple delimiter handling

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

Multiple delimiter handling, specified as the comma-separated pair consisting of 'MultipleDelimsAsOne' and either true or false. If true, textscan treats consecutive delimiters as a single delimiter. Repeated delimiters separated by white-space are also treated as a single delimiter. You must also specify the Delimiter option.

Example: 'MultipleDelimsAsOne',1

'ReturnOnError' - Behavior when textscan fails to read or convert

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

Behavior when textscan fails to read or convert, specified as the comma-separated pair consisting of 'ReturnOnError' and either true or false. If true, textscan terminates without an error and returns all fields read. If false, textscan terminates with an error and does not return an output cell array.

'TreatAsEmpty' - Strings to treat as empty value

string | cell array of strings

Strings to treat as empty values, specified as the comma-separated pair consisting of 'TreatAsEmpty' and a single string or cell array of strings. This option only applies to numeric fields.

'Whitespace' - White-space characters

' \b\t' (default) | string

White-space characters, specified as the comma-separated pair consisting of 'Whitespace' and a string of one or more characters. textscan adds a space character, char(32), to any specified

Whitespace, unless Whitespace is empty ('') and formatSpec includes any string conversion specifier.

When you specify one of the following escape sequences as any white-space character, textscan converts that sequence to the corresponding control character:

\b	Backspace
\n	Newline
\r	Carriage return
\t	Tab
\\	Backslash (\)

Output Arguments

C - File or string data

cell array

File or string data, returned as a cell array.

For each numeric conversion specifier in formatSpec, the textscan function returns a K-by-1 MATLAB numeric vector to the output cell array, C, where K is the number of times that textscan finds a field matching the specifier.

For each string conversion specifier in formatSpec, the textscan function returns a K-by-1 cell vector of strings, where K is the number of times that textscan finds a field matching the specifier. For each character conversion that includes a field width operator, textscan returns a K-by-M character array, where M is the field width.

position - File or string position

integer

File or string position at the end of the scan, returned as an integer of class double. For a file, ftell(fileID) would return the same value after calling textscan. For a string, position indicates how many characters textscan read.

Examples

Read a String

Read a string of floating-point numbers.

```
str = '0.41 8.24 3.57 6.24 9.27';

C = textscan(str,'%f');
```

The `formatSpec` string `'%f'` tells `textscan` to match each field in `str` to a double-precision floating-point number.

Display the contents of cell array `C`.

```
celldisp(C)
```

```
C{1} =

    0.4100
    8.2400
    3.5700
    6.2400
    9.2700
```

Read the same string, truncating each value to one decimal digit.

```
C = textscan(str,'%3.1f %*1d');
```

The specifier `%3.1f` indicates a field width of 3 digits and a precision of 1. `textscan` reads a total of 3 digits, including the decimal point and the 1 digit after the decimal point. The specifier, `%*1d`, tells `textscan` to skip the remaining digit.

Display the contents of cell array `C`.

```
celldisp(C)
```

```
C{1} =

    0.4000
```

```
    8.2000
    3.5000
    6.2000
    9.2000
```

Read Different Types of Data

Using a text editor, create a file `scan1.dat` that contains data in the following form:

```
09/12/2005 Level1 12.34 45 1.23e10 inf Nan Yes 5.1+3i
10/12/2005 Level2 23.54 60 9e19 -inf  0.001 No 2.2-.5i
11/12/2005 Level3 34.90 12 2e5   10  100   No 3.1+.1i
```

Open the file, and read each column with the appropriate conversion specifier.

```
fileID = fopen('scan1.dat');
C = textscan(fileID, '%s %s %f32 %d8 %u %f %f %s %f');
fclose(fileID);
celldisp(C)
```

```
C{1}{1} =

09/12/2005


C{1}{2} =

10/12/2005


C{1}{3} =

11/12/2005


C{2}{1} =
```

```
Level1


C{2}{2} =

Level2


C{2}{3} =

Level3


C{3} =

   12.3400
   23.5400
   34.9000



C{4} =

   45
   60
   12



C{5} =

  4294967295
  4294967295
      200000
```

```
C{6} =

   Inf
  -Inf
    10



C{7} =

        NaN
     0.0010
   100.0000



C{8}{1} =

Yes


C{8}{2} =

No


C{8}{3} =

No


C{9} =

   5.1000 + 3.0000i
   2.2000 - 0.5000i
   3.1000 + 0.1000i
```

`textscan` returns a 1-by-9 cell array `C`.

View the MATLAB data type of each of the cells in `C`.

```
C
```

```
C =

  Columns 1 through 5

    {3x1 cell}    {3x1 cell}    [3x1 single]    [3x1 int8]    [3x1 uint32

  Columns 6 through 9

    [3x1 double]    [3x1 double]    {3x1 cell}    [3x1 double]
```

For example, `C{1}` and `C{2}` are cell arrays. `C{5}` is of data type `uint32`, so the first two elements of `C{5}` are the maximum values for a 32-bit unsigned integer, or `intmax('uint32')`.

Remove a Literal String

Remove the text `'Level'` from each field in the second column of the data from the previous example.

Match the literal string in the `formatSpec` input.

```
fileID = fopen('scan1.dat');
C = textscan(fileID, '%s Level%d %f32 %d8 %u %f %f %s %f');
fclose(fileID);
C{2}
```

```
ans =

           1
           2
           3
```

View the MATLAB data type of the second cell in `C`.

```
class(C{2})

ans =

int32
```

The second cell of the 1-by-9 cell array, `C`, is now of data type `int32`.

Skip the Remainder of a Line

Read the first column of the file in the previous example into a cell array, skipping the rest of the line.

```
fileID = fopen('scan1.dat');
dates = textscan(fileID, '%s %*[^\n]');
fclose(fileID);
dates{1}

ans =

    '09/12/2005'
    '10/12/2005'
    '11/12/2005'
```

`textscan` returns a 1-by-1 cell array `dates`.

Specify Delimiter and Empty Value Conversion

Using a text editor, create a comma-delimited file, `data.csv`, that contains

```
1,  2,  3,  4,   ,  6
7,  8,  9,   , 11, 12
```

Read the file, converting empty cells to `-Inf`.

```
fileID = fopen('data.csv');
C = textscan(fileID, '%f %f %f %f %u8 %f','delimiter', ',', 'EmptyValu
fclose(fileID);
column4 = C{4}, column5 = C{5}
```

```
column4 =

     4
  -Inf


column5 =

    0
   11
```

textscan returns a 1-by-6 cell array, C. The textscan function converts the empty value in C{4} to -Inf, where C{4} is associated with a floating-point format. Because MATLAB represents unsigned integer -Inf as 0, textscan converts the empty value in C{5} to 0, and not -Inf.

Read Custom Empty Value Strings and Comments

Using a text editor, create a comma-delimited file, data2.csv, that contains the lines

```
abc, 2, NA, 3, 4
// Comment Here
def, na, 5, 6, 7
```

Designate the input that textscan should treat as comments or empty values.

```
fileID = fopen('data2.csv');
C = textscan(fileID, '%s %n %n %n %n', 'delimiter', ',', 'treatAsEmpty',
fclose(fileID);
celldisp(C)
```

```
C{1}{1} =
abc

C{1}{2} =
def
```

```
C{2} =
     2
   NaN

C{3} =
   NaN
     5

C{4} =
     3
     6

C{5} =
     4
     7
```

Treat Repeated Delimiters as One

Using a text editor, create a file, `data3.csv`, that contains

```
1,2,3,,4
5,6,7,,8
```

To treat the repeated commas as a single delimiter, use the `MultipleDelimsAsOne` parameter, and set the value to `1 (true)`.

```
fileID = fopen('data3.csv');
C = textscan(fileID, '%f %f %f %f', 'delimiter', ',', 'MultipleDelimsA
fclose(fileID);
celldisp(C)

C{1} =
     1
     5

C{2} =
     2
     6
```

```
C{3} =
     3
     7

C{4} =
     4
     8
```

Collect Numeric Data

Using a text editor, create a file, `grades.txt`, that contains:

```
Student_ID  | Test1  | Test2  | Test3
   1           91.5     89.2     77.3
   2           88.0     67.8     91.0
   3           76.3     78.1     92.5
   4           96.4     81.2     84.6
```

Read the column headers using the format `'%s'` four times.

```
fileID = fopen('grades.txt');

formatSpec = '%s';
N = 4;
C_text = textscan(fileID, formatSpec, N, 'delimiter', '|');
```

Read the numeric data in the file.

```
C_data0 = textscan(fileID, '%d %f %f %f')
```

```
C_data0 =
  [4x1 int32]    [4x1 double]    [4x1 double]    [4x1 double]
```

The default value for `CollectOutput` is 0 `(false)`, so `textscan` returns each column of the numeric data in a separate array.

Set `CollectOutput` to `1 (true)` to collect the consecutive columns of the same class into a single array.

```
frewind(fileID);

C_text = textscan(fileID, '%s', N, 'delimiter', '|');

C_data1 = textscan(fileID, '%d %f %f %f', 'CollectOutput', 1)
```

```
C_data1 =
    [4x1 int32]    [4x3 double]
```

The test scores, which are all `double`, are collected into a single 4-by-3 array.

Close the file, `grades.txt`.

```
fclose(fileID);
```

Read Nondefault Control Characters

Use `sprintf` to convert nondefault escape sequences in your data.

Create a string that includes a form feed character, `\f`. Then, to read the string using `textscan`, call `sprintf` to explicitly convert the form feed.

```
lyric = sprintf('Blackbird\fsinging\fin\fthe\fdead\fof\fnight');
C = textscan(lyric, '%s', 'delimiter', sprintf('\f'));
C{1}
```

```
ans =

    'Blackbird'
    'singing'
    'in'
    'the'
    'dead'
    'of'
    'night'
```

`textscan` returns a 1-by-1 cell array, `C`.

Resume a Text Scan of a String

Resume a scan of a string from a position other than the beginning.

If you resume a text scan of a string, `textscan` reads from the beginning of the string each time. To resume a scan from any other position in the string, use the two-output argument syntax in your initial call to `textscan`.

For example, create a string called `lyric`. Read the first word of the string, and then resume the scan.

```
lyric = 'Blackbird singing in the dead of night';
[firstword, pos] = textscan(lyric,'%9c', 1);
lastpart = textscan(lyric(pos+1:end), '%s');
```

Algorithms

`textscan` converts numeric fields to the specified output type according to MATLAB rules regarding overflow, truncation, and the use of `NaN`, `Inf`, and `-Inf`. For example, MATLAB represents an integer `NaN` as zero. If `textscan` finds an empty field associated with an integer format specifier (such as `%d` or `%u`), it returns the empty value as zero and not `NaN`.

`textscan` does not include leading white-space characters in the processing of any data fields. When processing numeric data, `textscan` also ignores trailing white space.

`textscan` imports any complex number as a whole into a complex numeric field, converting the real and imaginary parts to the specified numeric type (such as `%d` or `%f`). Valid forms for a complex number are:

`±<real>±<imag>i|j`	Example: `5.7-3.1i`
`±<imag>i|j`	Example: `-7j`

Do not include embedded white space in a complex number. `textscan` interprets embedded white space as a field delimiter.

See Also

`dlmread` | `fread` | `fscanf` | `importdata` | `load` | `uiimport` | `xlsread` | `fopen`

Related Examples

- "Import Data from a Nonrectangular Text File"
- "Import Large Text Files"
- "Access Data in a Cell Array"

Purpose

Wrapped string matrix for given uicontrol

Syntax

```
outstring = textwrap(h,instring)
outstring = textwrap(h,instring,columns)
[outstring,position] = textwrap(...)
```

Description

`outstring = textwrap(h,instring)` returns a wrapped string cell array, `outstring`, that fits inside the uicontrol with handle `h`. `instring` is a cell array, with each cell containing a single line of text. `outstring` is the wrapped string matrix in cell array format. Each cell of the input string is considered a paragraph.

`outstring = textwrap(h,instring,columns)` returns an `outstring` with each line wrapped at `columns` characters. Spaces are included in the character count.

`[outstring,position] = textwrap(...)` returns the recommended position of the uicontrol in the units of the uicontrol. `position` considers the extent of the multiline text in the x and y directions.

`textwrap` maintains the original line breaks in the input cell array and adds new ones. It can calculate uicontrol positions with any type of `Units`, including normalized units.

Tips

When programming a GUI, do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

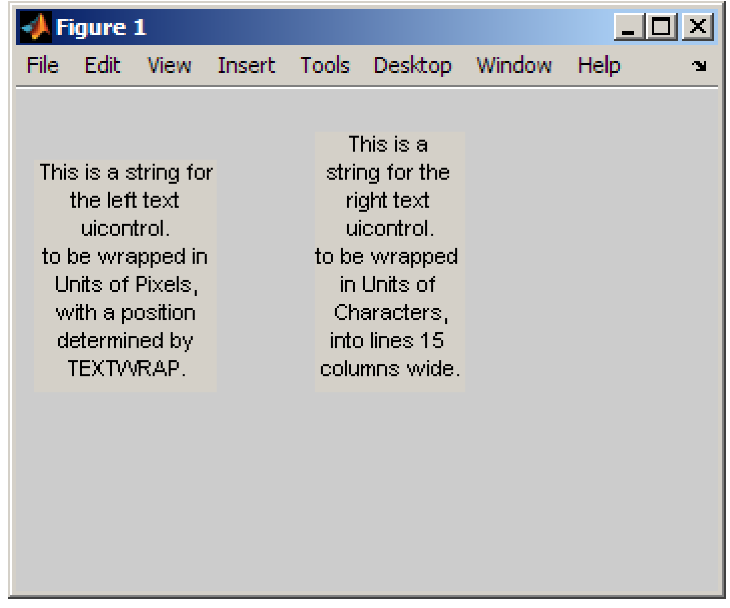
Examples

Place two text-wrapped strings in text uicontrols. The left one has a `Position` calculated by `textwrap` in `Units` of `pixels`; the right one's `Position` is calculated manually in `Units` of `characters`:

```
figure('Position',[560 528 350 250]);
% Make a text uicontrol to wrap in Units of Pixels
% Create it in Units of Pixels, 100 wide, 10 high
pos = [10 100 100 10];
ht = uicontrol('Style','Text','Position',pos);
```

```
string = {'This is a string for the left text uicontrol.',...
          'to be wrapped in Units of Pixels,',...
          'with a position determined by TEXTWRAP.'};
% Wrap string, also returning a new position for ht
[outstring,newpos] = textwrap(ht,string) %#ok<NOPRT>

outstring =
    'This is a string for'
    'the left text'
    'uicontrol.'
    'to be wrapped in'
    'Units of Pixels,'
    'with a position'
    'determined by'
    'TEXTWRAP.'

newpos =
    10   100    91   124

set(ht,'String',outstring,'Position',newpos)

% Make another text uicontrol to wrap to a column width of 15
colwidth = 15;
% Create it in Units of Pixels, 100 wide, 10 high
pos1 = [150 100 100 10];
ht1 = uicontrol('Style','Text','Position',pos1);
string1 = {'This is a string for the right text uicontrol.',...
          'to be wrapped in Units of Characters,',...
          'into lines 15 columns wide.'};
outstring1 = textwrap(ht1,string1,colwidth);
% Reset Units of ht1 to Characters to use the result
set(ht1,'Units','characters')
newpos1 = get(ht1,'Position')

newpos1 =
   29.8000    7.6154   20.0000    0.7692
```

```
% Set new Position in Characters to be specified colwidth
% with height the length of the outstring1 cell array + 1.
newpos1(3) = colwidth;
newpos1(4) = length(outstring1)+1

newpos1 =
   29.8000    7.6154   15.0000   10.0000

set(ht1,'String',outstring1,'Position',newpos1)
```



See Also

align | uicontrol

Purpose

Transpose-free quasi-minimal residual method

Syntax

```
x = tfqmr(A,b)
x = tfqmr(afun,b)
x = tfqmr(a,b,tol)
x = tfqmr(a,b,tol,maxit)
x = tfqmr(a,b,tol,maxit,m)
x = tfqmr(a,b,tol,maxit,m1,m2,x0)
[x,flag] = tfqmr(A,B,...)
[x,flag,relres] = tfqmr(A,b,...)
[x,flag,relres,y]y(A,b,...)
[x,flag,relres,iter,resvec] = tfqmr(A,b,...)
```

Description

`x = tfqmr(A,b)` attempts to solve the system of linear equations `A*x=b` for `x`. The `n`-by-`n` coefficient matrix `A` must be square and the right-hand side column vector `b` must have length `n`.

`x = tfqmr(afun,b)` accepts a function handle, `afun`, instead of the matrix `A`. The function, `afun(x)`, accepts a vector input `x` and returns the matrix-vector product `A*x`. In all of the following syntaxes, you can replace `A` by `afun`. "Parameterizing Functions" explains how to provide additional parameters to the function `afun`.

`x = tfqmr(a,b,tol)` specifies the tolerance of the method. If `tol` is `[]` then `tfqmr` uses the default, 1e-6.

`x = tfqmr(a,b,tol,maxit)` specifies the maximum number of iterations. If `maxit` is `[]` then `tfqmr` uses the default, `min(N,20)`.

`x = tfqmr(a,b,tol,maxit,m)` and `x = tfqmr(a,b,tol,maxit,m1,m2)` use preconditioners `m` or `m=m1*m2` and effectively solve the system `A*inv(M)*x = B` for `x`. If `M` is `[]` then a preconditioner is not applied. `M` may be a function handle `mfun` such that `mfun(x)` returns `m\x`.

`x = tfqmr(a,b,tol,maxit,m1,m2,x0)` specifies the initial guess. If `x0` is `[]` then `tfqmr` uses the default, an all zero vector.

`[x,flag] = tfqmr(A,B,...)` also returns a convergence flag:

Flag	Convergence
0	`tfqmr` converged to the desired tolerance `tol` within `maxit` iterations.
1	`tfqmr` iterated `maxit` times but did not converge.
2	Preconditioner `m` was ill-conditioned.
3	`tfqmr` stagnated. (Two consecutive iterates were the same.)
4	One of the scalar quantities calculated during `tfqmr` became too small or too large to continue computing.

`[x,flag,relres] = tfqmr(A,b,...)` also returns the relative residual `norm(b-A*x)/norm(b)`. If `flag` is 0, then `relres <= tol`.

`[x,flag,relres,y]y(A,b,...)` also returns the iteration number at which `x` was computed: `0 <= iter <= maxit`.

`[x,flag,relres,iter,resvec] = tfqmr(A,b,...)` also returns a vector of the residual norms at each iteration, including `norm(b-A*x0)`.

Examples

Example 1

This example shows how to use `tfqmr` with a matrix input and with a function input.

```
n = 100; on = ones(n,1);
A = spdiags([-2*on 4*on -on],-1:1,n,n);
b = sum(A,2);
tol = 1e-8;
maxit = 15;
M1 = spdiags([on/(-2) on],-1:0,n,n);
M2 = spdiags([4*on -on],0:1,n,n);
x = tfqmr(A,b,tol,maxit,M1,M2,[]);
```

You can also use a matrix-vector product function as input:

```
function y = afun(x,n)
```

```
y = 4 * x;
y(2:n) = y(2:n) - 2 * x(1:n-1);
y(1:n-1) = y(1:n-1) - x(2:n);
x1 = tfqmr(@(x)afun(x,n),b,tol,maxit,M1,M2);
```

If `applyOp` is a function suitable for use with `qmr`, it may be used with `tfqmr` by wrapping it in an anonymous function:

```
x1 = tfqmr(@(x)applyOp(x,'notransp'),b,tol,maxit,M1,M2);
```

Example 2

This example demonstrates the use of a preconditioner.

1

Load `A = west0479`, a real 479-by-479 nonsymmetric sparse matrix:

```
load west0479;
A = west0479;
```

2

Define `b` so that the true solution is a vector of all ones:

```
b = full(sum(A,2));
```

3

Set the tolerance and maximum number of iterations:

```
tol = 1e-12; maxit = 20;
```

4

Use `tfqmr` to find a solution at the requested tolerance and number of iterations:

```
[x0,fl0,rr0,it0,rv0] = tfqmr(A,b,tol,maxit);
```

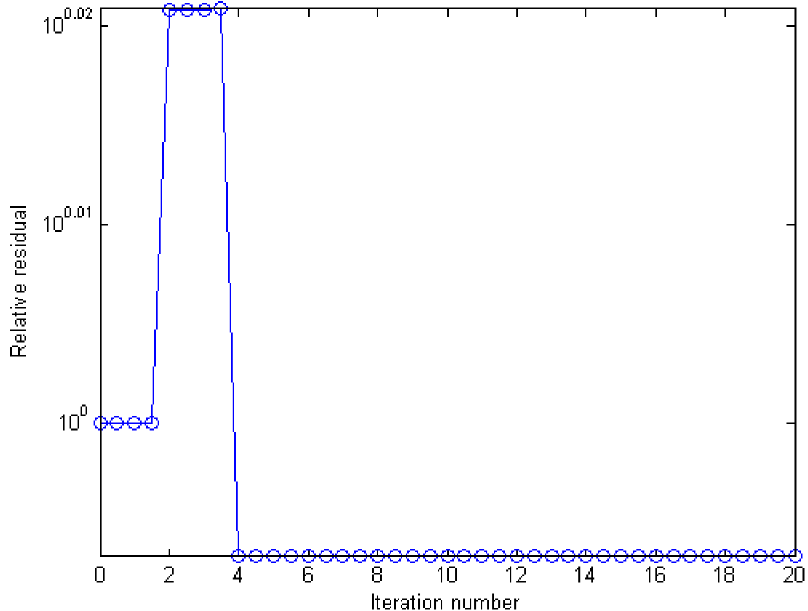
`fl0` is 1 because `tfqmr` does not converge to the requested tolerance `1e-12` within the requested 20 iterations. The seventeenth iterate is the

best approximate solution and is the one returned as indicated by `it0` = 17. MATLAB stores the residual history in `rv0`.

5

Plot the behavior of `tfqmr`:

```
semilogy(0:maxit,rv0/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



Note that like `bicgstab`, `tfqmr` keeps track of half iterations. The plot shows that the solution does not converge. You can use a preconditioner to improve the outcome.

6

Create the preconditioner with `ilu`, since the matrix `A` is nonsymmetric:

```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-5));
Error using ilu
There is a pivot equal to zero. Consider decreasing
the drop tolerance or consider using the 'udiag' option.
```

MATLABcannot construct the incomplete LU as it would result in a singular factor, which is useless as a preconditioner.

7

You can try again with a reduced drop tolerance, as indicated by the error message:

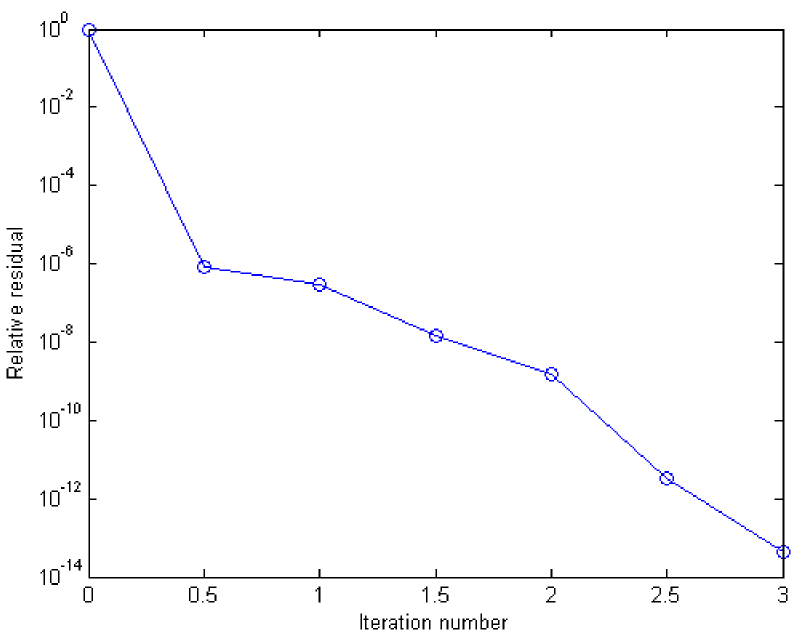
```
[L,U] = ilu(A,struct('type','ilutp','droptol',1e-6));
[x1,fl1,rr1,it1,rv1] = tfqmr(A,b,tol,maxit,L,U);
```

`fl1` is 0 because `tfqmr` drives the relative residual to `4.1410e-014` (the value of `rr1`). The relative residual is less than the prescribed tolerance of `1e-12` at the sixth iteration (the value of `it1`) when preconditioned by the incomplete LU factorization with a drop tolerance of `1e-6`. The output `rv1(1)` is `norm(b)`, and the output `rv1(7)` is `norm(b-A*x2)`.

8

You can follow the progress of `tfqmr` by plotting the relative residuals at each iteration starting from the initial estimate (iterate number 0):

```
semilogy(0:it1,rv1/norm(b),'-o');
xlabel('Iteration number');
ylabel('Relative residual');
```



See Also

qmr | bicg | bicgstab | bicgstabl | cgs | gmres | lsqr | luinc | minres | pcg | symmlq | mldivide | function_handle

Purpose

Issue exception and terminate function

Syntax

`throw(exception)`

Description

`throw(exception)` issues an exception based on the information contained in *`exception`*. The exception terminates the currently running function and returns control to its caller. The *`exception`* argument is scalar object of the `MException` class that contains information on the cause of the error and where it occurred. The `throw` function passes *`exception`* back to the caller of the currently running function. and eventually back to the Command Window when the program terminates. The exception is made available to any calling function by means of the `catch` function, and to the Command Window by means of the `MException.last` function.

Unlike `throwAsCaller` and `rethrow`, the `throw` function also sets the *stack* field of the *`exception`* to the location from which `throw` was called.

Tips

There are four ways to throw an exception in MATLAB (see the list below). Use the first of these when testing the outcome of some action for failure and reporting the failure to MATLAB. Use one of the remaining three techniques to throw an existing exception.

1. Test the result of some action taken by your program. If the result is found to be incorrect or unexpected, compose an appropriate message and message identifier, and pass these to MATLAB using the `error` function.

2. Reissue the original exception by throwing the initial exception unmodified. Use the `MException` `rethrow` method to do this.

3. Collect additional information on the cause of the error, store it in a new or modified exception, and issue a new exception based on that record. Use the `MException` `addCause` and `throw` methods to do this.

4. Make it appear that the error originated in the caller of the currently running function. Use the `MException` `throwAsCaller` method to do this.

Examples

Example 1

This example tests the output of function `evaluate_plots` and throws an exception if it is not acceptable:

```
[minval, maxval] = evaluate_plots(p24, p28, p41);
if minval < lower_bound || maxval > upper_bound
    exception = MException('VerifyOutput:OutOfBounds', ...
       'Results are outside the allowable limits');
    throw(exception);
end
```

Example 2

This example attempts to open a file in a folder that is not on the MATLAB path. It uses a nested `try-catch` block to give the user the opportunity to extend the path. If the file still cannot be found, the program issues an exception with the first error appended to the second using `addCause`:

```
function data = read_it(filename);
try
   % Attempt to open and read from a file.
   fid = fopen(filename, 'r');
   data = fread(fid);
catch exception1
   % If the error was caused by an invalid file ID, try
   % reading from another location.
   if strcmp(exception1.identifier, 'MATLAB:FileIO:InvalidFid')
      msg = sprintf( ...
         '\nCannot open file %s. Try another location?  ', ...
         filename);
      reply = input(msg, 's')
      if reply(1) == 'y'
          newFolder = input('Enter folder name:  ', 's');
```

```
        else
            throw(exception1);
        end
        oldpath = addpath(newFolder);
        try
           fid = fopen(filename, 'r');
           data = fread(fid);
        catch exception2
           exception3 = addCause(exception2, exception1)
           path(oldpath);
           throw(exception3);
        end
        path(oldpath);
    end
end
fclose(fid);

try
   d = read_it('anytextfile.txt');
catch exception
end

exception
exception =
 MException object with properties:

    identifier: 'MATLAB:FileIO:InvalidFid'
       message: 'Invalid file identifier.  Use fopen
                 to generate a valid file identifier.'
         stack: [1x1 struct]
         cause: {[1x1 MException]}

  Cannot open file anytextfile.txt. Try another location?y
Enter folder name:  xxxxxxx
Warning: Name is nonexistent or not a directory: xxxxxxx.
> In path at 110
  In addpath at 89
```

See Also

`try` | `catch` | `error` | `assert` | `MException` | `throwAsCaller(MException)` | `rethrow(MException)` | `addCause(MException)` | `getReport(MException)` | `last(MException)`

Purpose

Throw exception as if from calling function

Syntax

```
throwAsCaller(exception)
```

Description

`throwAsCaller(exception)` throws an exception from the currently running function based on the `exception` input, a scalar object of the `MException` class. The MATLAB software exits the currently running function and returns control to either the keyboard or an enclosing `catch` block in a calling function. Unlike the `throw` function, MATLAB omits the current stack frame from the `stack` field of the `MException`, thus making the exception look as if it is being thrown by the caller of the function.

In some cases, it is not relevant to show the person running your program the true location that generated an exception, but is better to point to the calling function where the problem really lies. You might also find `throwAsCaller` useful when you want to simplify the error display, or when you have code that you do not want made public.

Tips

There are four ways to throw an exception in MATLAB (see the list below). Use the first of these when testing the outcome of some action for failure and reporting the failure to MATLAB. Use one of the remaining three techniques to throw an existing exception.

1. Test the result of some action taken by your program. If the result is found to be incorrect or unexpected, compose an appropriate message and message identifier, and pass these to MATLAB using the `error` function.

2. Reissue the original exception by throwing the initial exception unmodified. Use the `MException` `rethrow` method to do this.

3. Collect additional information on the cause of the error, store it in a new or modified exception, and issue a new exception based on that record. Use the `MException` `addCause` and `throw` methods to do this.

4. Make it appear that the error originated in the caller of the currently running function. Use the `MException` `throwAsCaller` method to do this.

Examples

The function `klein_bottle`, in this example, generates a Klein Bottle figure by revolving the figure-eight curve defined by XYKLEIN. It defines a few variables and calls the function `draw_klein`, which executes three functions in a `try-catch` block. If there is an error, the `catch` block issues an exception using either `throw` or `throwAsCaller`:

```
function klein_bottle(pq)
ab = [0 2*pi];
rtr = [2 0.5 1];
box = [-3 3 -3 3 -2 2];
vue = [55 60];
draw_klein(ab, rtr, pq, box, vue)

function draw_klein(ab, rtr, pq, box, vue)
clf
try
   tube('xyklein',ab, rtr, pq, box, vue);
   shading interp
   colormap(pink);
catch exception
   throw(exception)
%   throwAsCaller(exception)
end
```

Call the `klein_bottle` function, passing a vector, and the function completes normally by drawing the figure.

```
klein_bottle([40 40])
```

Call the function again, this time passing a scalar value. Because the `catch` block issues the exception using `throw`, MATLAB displays error messages for line 16 of function `draw_klein`, and for line 6 of function `klein_bottle`:

```
klein_bottle(40)
Error using klein_bottle>draw_klein (line 16)
Attempted to access pq(2); index out of bounds because numel(pq)=1.

Error in klein_bottle (line 6)
draw_klein(ab, rtr, pq, box, vue)
```

Run the function again, this time changing the `klein_bottle.m` file so that the `catch` block uses `throwAsCaller` instead of `throw`. This time, MATLAB only displays the error at line 6 of the main program:

```
klein_bottle(40)
Error using klein_bottle (line 6)
Attempted to access pq(2); index out of bounds because numel(pq)=1.
```

See Also

`try` | `catch` | `error` | `assert` | `MException` | `throw(MException)` | `rethrow(MException)` | `addCause(MException)` | `getReport(MException)` | `last(MException)`

Purpose

Start clock to measure performance

Syntax

```
tic
ticID = tic
```

Description

`tic` starts a stopwatch timer. Stop the timer with the `toc` function.

`ticID = tic` stores an identifier for the `tic` command, so that you can nest timing operations.

Tips

- Consecutive `tic` commands overwrite the internally recorded starting time.
- The `clear` function does not reset the starting time recorded by a `tic` command.

Output Arguments

ticID

Value to use as an input argument for a subsequent call to `toc`. The value has no independent meaning.

Examples

Measure how the time required to solve a linear system varies with the order of a matrix:

```
t = zeros(1,100);
for n = 1:100
    A = rand(n,n);
    b = rand(n,1);
    tic;
    x = A\b;
    t(n) = toc;
end
plot(t)
```

Measure the minimum and average time to compute a summation of Bessel functions:

```
REPS = 1000;   minTime = Inf;   nsum = 10;
tic;

for i=1:REPS
   tStart = tic;   total = 0;
   for j=1:nsum
      total = total + besselj(j,REPS);
   end

   tElapsed = toc(tStart);
   minTime = min(tElapsed, minTime);
end
averageTime = toc/REPS;
```

See Also

clock | cputime | etime | profile | toc

Purpose

Stop clock to measure performance

Syntax

```
toc
toc(ticID)
elapsedTime = toc
elapsedTime = toc(ticID)
```

Description

`toc` stops a stopwatch timer started by the `tic` function, and displays the time elapsed in seconds.

`toc(ticID)` displays the time elapsed since the `tic` command corresponding to `ticID`.

`elapsedTime = toc` stores the elapsed time in a variable.

`elapsedTime = toc(ticID)` stores in a variable the time elapsed since the `tic` command corresponding to `ticID`.

Tips

- Consecutive `toc` commands return the increasing time that has elapsed since the most recent `tic`. Therefore, you can take multiple measurements from a single point in time.
- The `toc` function displays the time elapsed unless you specify an output argument.

Input Arguments

ticID

Identifier generated by a previous call to `tic` with an output argument.

Output Arguments

elapsedTime

Scalar `double` representing the time elapsed between `tic` and `toc` commands, in seconds.

Examples

Measure how the time required to solve a linear system varies with the order of a matrix:

```
t = zeros(1,100);
for n = 1:100
```

```
    A = rand(n,n);
    b = rand(n,1);
    tic;
    x = A\b;
    t(n) = toc;
end
plot(t)
```

Measure the minimum and average time to compute a summation of Bessel functions:

```
REPS = 1000;   minTime = Inf;   nsum = 10;
tic;

for i=1:REPS
   tStart = tic;   total = 0;
   for j=1:nsum
      total = total + besselj(j,REPS);
   end

   tElapsed = toc(tStart);
   minTime = min(tElapsed, minTime);
end
averageTime = toc/REPS;
```

See Also

`clock` | `cputime` | `etime` | `profile` | `tic`

Purpose

MATLAB Gateway to LibTIFF library routines

Description

The `Tiff` class represents a connection to a Tagged Image File Format (TIFF) file and provides access to many of the capabilities of the LibTIFF library. Use the methods of the `Tiff` object to call routines in the LibTIFF library. While you can use the `imread` and `imwrite` functions to read and write TIFF files, the `Tiff` class offers capabilities that these functions don't provide, such as reading subimages, writing tiles and strips of image data, and modifying individual TIFF tags.

In most cases, the syntax of the `Tiff` method is similar to the syntax of the corresponding LibTIFF library function. To get the most out of the `Tiff` object, you must be familiar with the LibTIFF API, as well as the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

MATLAB supports LibTIFF version 4.0.0.

For copyright information, see the `libtiffcopyright.txt` file.

Construction

`obj = Tiff(filename,mode)` creates a `Tiff` object associated with the TIFF file `filename`. `mode` specifies the type of access to the file.

A TIFF file is made up of one or more image file directories (IFDs). An IFD contains image data and associated metadata. IFDs can also contain subIFDs which also contain image data and metadata. When you open a TIFF file for reading, the Tiff object makes the first IFD in the file the *current* IFD. `Tiff` methods operate on the current IFD. You can use `Tiff` object methods to navigate among the IFDs and the subIFDs in a TIFF file.

When you open a TIFF file for writing or appending, the `Tiff` object automatically creates a IFD in the file for writing subsequent data. This IFD has all the default values specified in TIFF Revision 6.0.

When creating a new TIFF file, before writing any image to the file, you must create certain required fields (tags) in the file. These tags include `ImageWidth`, `ImageHeight`, `BitsPerSample`, `SamplesPerPixel`, `Compression`, `PlanarConfiguration`, and `Photometric`. If the image data has a stripped layout, the IFD contains the `RowsPerStrip` tag. If

the image data has a tiled layout, the IFD contains the `TileWidth` and `TileHeight` tags. Use the `setTag` method to define values for these tags.

Input Arguments

filename

Text string specifying name of file.

mode

One of the following text strings specifying the type of access to the TIFF file.

Supported Values

Parameter	Description
`'r'`	Open file for reading (default)
`'w'`	Open file for writing; discard existing contents
`'w8'`	Open file for writing a BigTIFF file; discard existing contents
`'a'`	Open or create file for writing; append data to end of file.
`'r+'`	Open (do not create) file for reading and writing

Properties

Compression

Specify scheme used to compress image data

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

Supported Values

None
CCITTRLE (Read-only)
CCITTFax3
CCITTFax4
LZW
JPEG
CCITTRLEW (Read-only)
PackBits
SGILog
SGILog24
Deflate
AdobeDeflate (Same as deflate)

Example: Set the `Compression` tag to the value `JPEG`. Note how you use the property to specify the value.

```
tiffobj.setTag('Compression', Tiff.Compression.JPEG);
```

ExtraSamples

Describe extra components

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

This field is required if there are extra channels in addition to the usual colormetric channels.

Supported Values

Value of `ExtraSamples`	Description
`Unspecified`	Unspecified data
`AssociatedAlpha`	Associated alpha (pre-multiplied)
`UnassociatedAlpha`	Unassociated alpha data

Example: Set the `ExtraSamples` tag to the value `AssociatedAlpha`. Note how you use the property to specify the value.

```
tiffobj.setTag('ExtraSamples', Tiff.ExtraSamples.AssociatedAlpha)
```

See Also "Specify Tiff object properties and describe alpha channel" on page 1-5616

Group3Options

Options for Group 3 Fax Compression

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

This property is also referred to as `Fax3` and `T4Options`. This value is a bit mask controlled by the first three bits.

Supported Values

Supported values include the following.

Encoding2D	Bit 0 is 1. This specifies two-dimensional coding. If more than one strip is specified, each strip must begin with a one-dimensionally coded line. That is, RowsPerStrip should be a multiple of Parameter K, as documented in the CCITT specification.
Uncompressed	Bit 1 is 1. This specifies an uncompressed mode when encoding.
FillBits	Bit 2 is 1. Fill bits are added as necessary before EOL codes such that EOL always ends on a byte boundary. This ensures an EOL-sequence of 1 byte preceded by a zero nibble, for example, xxxx-0000 0000-0001.

Example:

```
mask = bitor(Tiff.Group3Options.Encoding2D,Tiff.Group3Options.Uncomp
tiffobj.setTag('Group3Options',mask);
```

InkSet

Specify set of inks used in separated image

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the setTag method. For more clarification, see the example.

In this context, separated refers to photometric interpretation, not the planar configuration.

Supported Values

`CMYK`	Order of components: cyan, magenta, yellow, black. Usually, a value of 0 represents 0% ink coverage and a value of 255 represents 100% ink coverage for that component, but consult the TIFF specification for `DotRange`. When you specify CMYK, do not set the `InkNames` tag.
`MultiInk`	Any ordering other than CMYK. Consult the TIFF specification for `InkNames` field for a description of the inks used.

Example:

```
tiffobj.setTag('InkSet', Tiff.InkSet.CMYK);
```

JPEGColorMode

Specify control of YCbCr/RGB conversion

Use these values only when the photometric interpretation is YCbCr.

This property should not be used for the purpose of reading YCbCr imagery as RGB. In this case, use the RGBA interface provided by the `readRGBAImage`, `readRGBAStrip`, and `readRGBATile` methods.

Supported Values

`Raw` (default)	Keep input as separate Y, Cb, and Cr matrices.
`RGB`	Convert RGB input to YCbCr.

Example:

```
tiffobj.setTag('JPEGColorMode',Tiff/JPEGColorMode.RGB);
```

See also “Create YCbCr/JPEG image from RGB data” on page 1-5616

Orientation

Specify visual orientation of the image data.

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

The first row represents the top of the image, and the first column represents the left side. Support for this tag is for informational purposes only, and it does not affect how MATLAB reads or writes the image data.

Supported Values

TopLeft
TopRight
BottomRight
BottomLeft
LeftTop
RightTop
RightBottom
LeftBottom

Example:

```
tiffobj.setTag('Orientation', Tiff.Orientation.TopRight);
```

Photometric

Specify color space of image data

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

Supported Values

MinIsWhite
MinIsBlack
RGB
Palette
Mask
Separated (CMYK)
YCbCr
CIELab
ICCLab
ITULab
LogL
LogLUV
CFA
LinearRaw

Example:

```
tiffobj.setTag('Photometric', Tiff.Photometric.RGB);
```

PlanarConfiguration

Specifies how image data components are stored on disk

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

Supported Values

`Chunky`	Store component values for each pixel contiguously. For example, in the case of RGB data, the first three pixels would be stored in the file as RGBRGBRGB etc. Almost all TIFF images have contiguous planar configurations.
`Separate`	Store component values for each pixel separately. For example, in the case of RGB data, the red component would be stored separately in the file from the green and blue components.

Example:

```
tiffobj.setTag('PlanarConfiguration', Tiff.PlanarConfiguration.Chunk
```

ResolutionUnit

Specify unit of measure used to interpret the `XResolution` and `YResolution` tags.

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

Supported Values

`None` (default)
`Inch`
`Centimeter`

Example: Set `ResolutionUnit` tag to the value `Inch`. Then, setting `XResolution` tag to 300 means pixels per inch.

```
tiffObj.setTag('ResolutionUnit', Tiff.ResolutionUnit.Inch);
tiffObj.setTag('XResolution', 300);
```

```
tiffObj.setTag('YResolution', 300);
```

SampleFormat

Specify how to interpret each pixel sample

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

Supported Values

Supported Values
`UInt` (default)
`Int`
`IEEEFP`
`Void`
`ComplexInt`
`ComplexIEEEFP`

Example:

```
tiffobj.setTag('SampleFormat', Tiff.SampleFormat.IEEEFP);
```

SGILogDataFmt

Specify control of client data for SGILog codec

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

These enumerated values should only be used when the photometric interpretation value is `LogL` or `LogLUV`. The `BitsPerSample`, `SamplesPerPixel`, and `SampleFormat` tags should not be set if the image type is `LogL` or `LogLuv`. The choice of `SGILogDataFmt` will set these tags automatically. The `Float` and `Bits8` settings imply a `SamplesPerPixel` value of 3 for `LogLUV` images, but only 1 for `LogL` images.

Supported Values

Float	Single precision samples
Bits8	uint8 samples (read only)

This tag can be set only once per instance of a LogL/LogLuv Tiff image object instance.

Example:

```
tiffobj = Tiff('example.tif','r');
tiffobj.setDirectory(3); % image three is a LogLuv image
tiffobj.setTag('SGILogDataFmt', Tiff.SGILogDataFmt.Float);
imdata = tiffobj.read();
```

SubFileType

Specify type of image

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the setTag method. For more clarification, see the example.

SubFileType is a bitmask that indicates the type of the image.

Supported Values

Default	Default value for single image file or first image.
ReducedImage	The current image is a thumbnail or reduced-resolution image that typically would be found in a sub-IFD.
Page	The image is a single image of a multi-image (or multipage) file.
Mask	The image is a transparency mask for another image in the file. The photometric interpretation value must be Photometric.Mask.

Example:

```
tiffobj.setTag('SubFileType', Tiff.SubFileType.Mask);
```

TagID

List of recognized TIFF tag names with their ID numbers

This property identifies all the supported TIFF tags with their ID numbers. Use this property to specify a tag when using the `setTag` method. For example, `Tiff.TagID.ImageWidth` returns the ID of the `ImageWidth` tag. To get a list of the names of supported tags, use the `getTagNames` method.

Example:

```
tiffobj.setTag(Tiff.TagID.ImageWidth, 300);
```

Thresholding

Specifies technique used to convert from gray to black and white pixels.

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

Supported Values

Supported Values
`BiLevel` (default)
`HalfTone`
`ErrorDiffuse`

Example:

```
tiffobj.setTag('Thresholding', Tiff.Thresholding.HalfTone);
```

YCbCrPositioning

Specify relative positioning of chrominance samples

This property defines all the supported values for this tag. You use this property to specify the value you want to assign to the tag, using the `setTag` method. For more clarification, see the example.

This property identifies all supported values for the `YCbCrPositioning` tag.

Supported Values

`Centered`	Specify for compatibility with industry standards such as PostScript Level 2
`Cosited`	Specify for compatibility with most digital video standards such as CCIR Recommendation 601-1.

Example:

```
tiffobj.setTag('YCbCrPositioning', Tiff.YCbCrPositioning.Centered);
```

Methods

close	Close Tiff object
computeStrip	Index number of strip containing specified coordinate
computeTile	Index number of tile containing specified coordinates
currentDirectory	Index of current IFD
getTag	Value of specified tag
getTagNames	List of recognized TIFF tags
getVersion	LibTIFF library version
isTiled	Determine if tiled image
lastDirectory	Determine if current IFD is last in file
nextDirectory	Make next IFD current IFD

numberOfStrips	Total number of strips in image
numberOfTiles	Total number of tiles in image
read	Read entire image
readEncodedStrip	Read data from specified strip
readEncodedTile	Read data from specified tile
readRGBAImage	Read image using RGBA interface
readRGBAStrip	Read strip data using RGBA interface
readRGBATile	Read tile data using RGBA interface
rewriteDirectory	Write modified metadata to existing IFD
setDirectory	Make specified IFD current IFD
setSubDirectory	Make subIFD specified by byte offset current IFD
setTag	Set value of tag
write	Write entire image
writeDirectory	Create new IFD and make it current IFD
writeEncodedStrip	Write data to specified strip
writeEncodedTile	Write data to specified tile

Examples

Create New TIFF File Using `Tiff` object

Create a new file called `myfile.tif`. To run this example, your directory must be writable.

```
t = Tiff('myfile.tif', 'w');
```

Close the Tiff object.

```
t.close();
```

Specify Tiff object properties and describe alpha channel

For this example, create an array of data, `data`, that contains colormetric channels and an alpha channel.

```
rgb = imread('example.tif');
numrows = size(rgb,1);
numcols = size(rgb,2);
alpha = 255*ones([numrows numcols], 'uint8');
data = cat(3,rgb,alpha);
```

Create a `Tiff` object, `t`, and set the object properties. Set the value of the `ExtraSamples` tag because the data contains the alpha channel in addition to the colormetric channels.

```
t = Tiff('myfile.tif','w');
t.setTag('Photometric',Tiff.Photometric.RGB);
t.setTag('Compression',Tiff.Compression.None);
t.setTag('BitsPerSample',8);
t.setTag('SamplesPerPixel',4);
t.setTag('SampleFormat',Tiff.SampleFormat.UInt);
t.setTag('ExtraSamples',Tiff/ExtraSamples.Unspecified);
t.setTag('ImageLength',numrows);
t.setTag('ImageWidth',numcols);
t.setTag('TileLength',32);
t.setTag('TileWidth',32);
t.setTag('PlanarConfiguration',Tiff.PlanarConfiguration.Chunky);
```

Write the data to the `Tiff` object.

```
t.write(data);
t.close();
```

Create YCbCr/JPEG image from RGB data

Get RGB data.

```
rgb = imread('example.tif');
```

Create a `Tiff` object, `t`, and set the object properties. Specify RGB input data using the `JPEGColorMode` property.

```
t = Tiff('myfile.tif','w');
t.setTag('Photometric',Tiff.Photometric.YCbCr);
t.setTag('Compression',Tiff.Compression.JPEG);
t.setTag('YCbCrSubSampling',[2 2]);
t.setTag('BitsPerSample',8);
t.setTag('SamplesPerPixel',3);
t.setTag('SampleFormat',Tiff.SampleFormat.UInt);
t.setTag('ImageLength',size(rgb,1));
t.setTag('ImageWidth',size(rgb,2));
t.setTag('TileLength',32);
t.setTag('TileWidth',32);
t.setTag('PlanarConfiguration',Tiff.PlanarConfiguration.Chunky);
t.setTag('JPEGColorMode',Tiff/JPEGColorMode.RGB);
t.setTag('JPEGQuality',75);
```

Write the data to the `Tiff` object.

```
t.write(rgb);
t.close();
```

See Also

`imread` | `imwrite` | `imfinfo`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

Purpose

Construct timer object

Syntax

```
T = timer
T = timer('PropertyName1', PropertyValue1, 'PropertyName2',
   PropertyValue2,...)
```

Description

`T = timer` constructs a timer object with default attributes.

`T = timer('PropertyName1', PropertyValue1, 'PropertyName2', PropertyValue2,...)` constructs a timer object in which the given property name/value pairs are set on the object. See “Timer Object Properties” on page 1-5618 for a list of all the properties supported by the timer object.

Note that the property name/property value pairs can be in any format supported by the `set` function, i.e., property/value string pairs, structures, and property/value cell array pairs.

Note The `timer` object is subject to the limitations of your hardware, operating system, and software. It is not intended to be used for real-time applications.

Timer Object Properties

The timer object supports the following properties that control its attributes. The table includes information about the data type of each property and its default value.

To view the value of the properties of a particular timer object, use the `get(timer)` function. To set the value of the properties of a timer object, use the `set(timer)` function.

Property Name	Property Description	Data Types, Values, Defaults, Access	
AveragePeriod	Average time between TimerFcn executions since the timer started. Note: Value is NaN until timer executes two timer callbacks.	Data type	double
		Default	NaN
		Read only	Always
BusyMode	Action taken when a timer has to execute TimerFcn before the completion of previous execution of TimerFcn. 'drop' — Skip execution when there is a function already pending in the queue. If the queue is empty, add the function to the queue and execute at the next opportunity. 'error' — Call the error handling function specified by ErrorFcn when there is already a function in the queue. 'queue' — Execute function at next opportunity. See "Handling Callback Function Queuing Conflicts" for more information.	Data type	Enumerated string
		Values	'drop' 'error' 'queue'
		Default	'drop'
		Read only	While Running = 'on'

Property Name	Property Description	Data Types, Values, Defaults, Access	
ErrorFcn	Function that the timer executes when an error occurs. This function executes before the StopFcn. See "Creating Callback Functions" for more information.	Data type	Text string, function handle, or cell array
		Default	None
		Read only	Never
ExecutionMode	Determines how the timer object schedules timer events. See "Timer Object Execution Modes" for more information.	Data type	Enumerated string
		Values	'singleShot' 'fixedDelay' 'fixedRate' 'fixedSpacing'
		Default	'singleShot'
		Read only	While Running = 'on'
InstantPeriod	The time between the last two executions of TimerFcn.	Data type	double
		Default	NaN
		Read only	Always

Property Name	Property Description	Data Types, Values, Defaults, Access	
`Name`	User-supplied name.	Data type	Text string
		Default	`'timer-`*`i`*`'`, where *i* is a number indicating the *i*th timer object created this session. To reset *i* to 1, execute the `clear classes` command.
		Read only	Never
`ObjectVisibility`	Provides a way for application developers to prevent end-user access to the timer objects created by their application. The `timerfind` function does not return an object whose `ObjectVisibility` property is set to `'off'`. Objects that are not visible are still valid. If you have access to the object (for example, from within the file that created it), you can set its properties.	Data type	Enumerated string
		Values	`'off'` `'on'`
		Default	`'on'`
		Read only	Never
`Period`	Specifies the delay, in seconds, between executions of `TimerFcn`.	Data type	`double`
		Value	Any number >= 0.001
		Default	1.0
		Read only	While `Running` = `'on'`

Property Name	Property Description	Data Types, Values, Defaults, Access	
`Running`	Indicates whether the timer is currently executing.	Data type	Enumerated string
		Values	`'off'` `'on'`
		Default	`'off'`
		Read only	Always
`StartDelay`	Specifies the delay, in seconds, between the start of the timer and the first execution of the function specified in `TimerFcn`.	Data type	`double`
		Values	Any number >= 0
		Default	0
		Read only	While `Running = 'on'`
`StartFcn`	Function the timer calls when it starts. See "Creating Callback Functions" for more information.	Data type	Text string, function handle, or cell array
		Default	None
		Read only	Never

Property Name	Property Description	Data Types, Values, Defaults, Access	
`StopFcn`	Function the timer calls when it stops. The timer stops when • You call the timer `stop` function • The timer finishes executing `TimerFcn`, i.e., the value of `TasksExecuted` reaches the limit set by `TasksToExecute`. • An error occurs (The `ErrorFcn` is called first, followed by the `StopFcn`.) See "Creating Callback Functions" for more information.	Date type	Text string, function handle, or cell array
		Default	None
		Read only	Never
`Tag`	User supplied label.	Data type	Text string
		Default	Empty string ('')
		Read only	Never

Property Name	Property Description	Data Types, Values, Defaults, Access	
`TasksToExecute`	Specifies the number of times the timer should execute the function specified in the `TimerFcn` property.	Data type	`double`
		Values	Any number > 0
		Default	`Inf`
		Read only	Never
`TasksExecuted`	The number of times the timer has called `TimerFcn` since the timer was started.	Data type	`double`
		Values	Any number >= 0
		Default	0
		Read only	Always
`TimerFcn`	Timer callback function. See “Creating Callback Functions” for more information.	Data type	Text string, function handle, or cell array
		Default	None
		Read only	Never
`Type`	Identifies the object type.	Data type	Text string
		Values	`'timer'`
		Read only	Always
`UserData`	User-supplied data.	Data type	User-defined
		Default	`[]`
		Read only	Never

Examples

This example constructs a timer object with a timer callback function handle, `mycallback`, and a 10 second interval.

```
t = timer('TimerFcn',@mycallback, 'Period', 10.0);
```

See Also

`delete(timer)` | `disp(timer)` | `get(timer)` | `isvalid(timer)` | `set(timer)` | `start` | `startat` | `stop` | `timerfind` | `timerfindall` | `wait`

timerfind

Purpose

Find timer objects

Syntax

```
out = timerfind
out = timerfind('P1', V1, 'P2', V2,...)
out = timerfind(S)
out = timerfind(obj, 'P1', V1, 'P2', V2,...)
```

Description

`out = timerfind` returns an array, `out`, of all the timer objects that exist in memory.

`out = timerfind('P1', V1, 'P2', V2,...)` returns an array, `out`, of timer objects whose property values match those passed as parameter/value pairs, `P1`, `V1`, `P2`, `V2`. Parameter/value pairs may be specified as a cell array.

`out = timerfind(S)` returns an array, `out`, of timer objects whose property values match those defined in the structure, `S`. The field names of `S` are timer object property names and the field values are the corresponding property values.

`out = timerfind(obj, 'P1', V1, 'P2', V2,...)` restricts the search for matching parameter/value pairs to the timer objects listed in `obj`. `obj` can be an array of timer objects.

Note When specifying parameter/value pairs, you can use any mixture of strings, structures, and cell arrays in the same call to `timerfind`.

Note that, for most properties, `timerfind` performs case-sensitive searches of property values. For example, if the value of an object's `Name` property is `'MyObject'`, `timerfind` will not find a match if you specify `'myobject'`. Use the `get` function to determine the exact format of a property value. However, properties that have an enumerated list of possible values are not case sensitive. For example, `timerfind` will find an object with an `ExecutionMode` property value of `'singleShot'` or `'singleshot'`.

Examples

These examples use `timerfind` to find timer objects with the specified property values.

```
t1 = timer('Tag', 'broadcastProgress', 'Period', 5);
t2 = timer('Tag', 'displayProgress');
out1 = timerfind('Tag', 'displayProgress')
out2 = timerfind({'Period', 'Tag'}, {5, 'broadcastProgress'})
```

See Also

`get(timer)` | `timer` | `timerfindall`

timerfindall

Purpose

Find timer objects, including invisible objects

Syntax

```
out = timerfindall
out = timerfindall('P1', V1, 'P2', V2,...)
out = timerfindall(S)
out = timerfindall(obj, 'P1', V1, 'P2', V2,...)
```

Description

`out = timerfindall` returns an array, `out`, containing all the timer objects that exist in memory, regardless of the value of the object's `ObjectVisibility` property.

`out = timerfindall('P1', V1, 'P2', V2,...)` returns an array, `out`, of timer objects whose property values match those passed as parameter/value pairs, `P1`, `V1`, `P2`, `V2`. Parameter/value pairs may be specified as a cell array.

`out = timerfindall(S)` returns an array, `out`, of timer objects whose property values match those defined in the structure, `S`. The field names of `S` are timer object property names and the field values are the corresponding property values.

`out = timerfindall(obj, 'P1', V1, 'P2', V2,...)` restricts the search for matching parameter/value pairs to the timer objects listed in `obj`. `obj` can be an array of timer objects.

Note When specifying parameter/value pairs, you can use any mixture of strings, structures, and cell arrays in the same call to `timerfindall`.

Note that, for most properties, `timerfindall` performs case-sensitive searches of property values. For example, if the value of an object's `Name` property is `'MyObject'`, `timerfindall` will not find a match if you specify `'myobject'`. Use the `get` function to determine the exact format of a property value. However, properties that have an enumerated list of possible values are not case sensitive. For example, `timerfindall` will find an object with an `ExecutionMode` property value of `'singleShot'` or `'singleshot'`.

Examples

Create several timer objects.

```
t1 = timer;
t2 = timer;
t3 = timer;
```

Set the `ObjectVisibility` property of one of the objects to `'off'`.

```
t2.ObjectVisibility = 'off';
```

Use `timerfind` to get a listing of all the timer objects in memory. Note that the listing does not include the timer object (`timer-2`) whose `ObjectVisibility` property is set to `'off'`.

```
timerfind

Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:   Name:
   1       singleShot      1        ''          timer-1
   2       singleShot      1        ''          timer-3
```

Use `timerfindall` to get a listing of all the timer objects in memory. This listing includes the timer object whose `ObjectVisibility` property is set to `'off'`.

```
timerfindall

Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:   Name:
   1       singleShot      1        ''          timer-1
   2       singleShot      1        ''          timer-2
   3       singleShot      1        ''          timer-3
```

See Also

`get(timer)` | `timer` | `timerfind`

times

Purpose

Array multiply

Syntax

```
c = a.*b
c = times(a,b)
```

Description

`c = a.*b` multiplies arrays `a` and `b` element-by-element and returns the result in `c`. Inputs `a` and `b` must have the same size unless one is a scalar.

`c = times(a,b)` is called for the syntax `a.*b` when `a` or `b` is an object.

See Also

`mtimes`

Purpose

Add title to current axes

Syntax

```
title(str)
title(str,Name,Value)

title(axes_handle,___)

h = title(___)
```

Description

`title(str)` adds the title consisting of a string, `str`, at the top and in the center of the current axes. Each axes graphics object has one predefined title.

`title(str,Name,Value)` additionally specifies the title properties using one or more `Name,Value` pair arguments.

`title(axes_handle,___)` adds the title to the axes specified by `axes_handle`. This syntax allows you to specify the axes to which to add a title. `axes_handle` can precede any of the input argument combinations in the previous syntaxes.

`h = title(___)` returns the handle to the text object used as the title. The handle is useful when making future modifications to the title.

Input Arguments

str - Text to display as title

string

Text to display as a title, specified as a string. You also can specify the name of a function that returns a string.

Example: `'myTitle'`

Example: `date`

Note The words `default`, `factory`, and `remove` are reserved words that will not appear in a title when quoted as a normal string. To display any of these words individually, type `'\reserved_word'` instead of `'reserved_word'`.

axes_handle - Axes handle

Axes handle, which is the reference to an axes object. Use the `gca` function to get the handle to the current axes, for example, `axes_handle = gca;`.

Name-Value Pair Arguments

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

Example: `'Color','red','FontSize',14` adds a title with red, 14-point font.

In addition to the following, you can specify other text object properties using `Name,Value` pair arguments. See Text Properties.

'Color' - Text color

`[0 0 0]` (black) (default) | 3-element RGB vector | string

Text color, specified as the comma-separated pair consisting of `'Color'` and a 3-element RGB vector or a string containing the short or long name of the color. The RGB vector is a 3-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0 1]`.

The following table lists the predefined colors and their RGB equivalents.

RGB Value	Short Name	Long Name
[1 1 0]	y	yellow
[1 0 1]	m	magenta
[0 1 1]	c	cyan
[1 0 0]	r	red
[0 1 0]	g	green
[0 0 1]	b	blue
[1 1 1]	w	white
[0 0 0]	k	black

Example: `'Color',[0 1 0]`

Example: `'Color','green'`

'FontAngle' - Character slant

`'normal'` (default) | `'italic'` | `'oblique'`

Character slant, specified as the comma-separated pair consisting of `'FontAngle'` and one of these values: `'normal'`, `'italic'`, or `'oblique'`. MATLAB uses the `FontAngle` property to select a font from those available on your particular system. Generally, setting this property to `italic` or `oblique` selects a slanted font.

Example: `'FontAngle','italic'`

'FontName' - Font name

`'Helvetica'` (default) | string | `'FixedWidth'`

Font name, specified as the comma-separated pair consisting of `'FontName'` and a string. The string specifies the name of the font to use for the text object. To display and print properly, this must be a font that your system supports.

To use a fixed-width font that looks good in any locale, use the case-sensitive string `'FixedWidth'`. This eliminates the need to

hard-code the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding.

Example: `'FontName','Courier'`

'FontSize' - Font size

10 points (default) | scalar

Font size, specified as the comma-separated pair consisting of `'FontSize'` and a scalar in units determined by the `FontUnits` property. The default value for `FontUnits` is `points`.

Example: `'FontSize',12.5`

'FontUnits' - Font size units

`'points'` (default) | `'normalized'` | `'inches'` | `'centimeters'` | `'pixels'`

Font size units, specified as the comma-separated pair consisting of `'FontUnits'` and one of the following strings:

- `'points'`
- `'normalized'`
- `'inches'`
- `'centimeters'`
- `'pixels'`

When the value of `FontUnits` is `'normalized'`, MATLAB interprets the value of `FontSize` as a fraction of the height of the parent axes. When you resize the axes, MATLAB modifies the screen `FontSize` accordingly. `points`, `inches`, `centimeters`, and `pixels` are absolute units. 1 point = $^1/_{72}$ inch

---

Note When setting both the `FontSize` and the `FontUnits`, you must set the `FontUnits` property first so that MATLAB can correctly interpret the specified `FontSize`. For example, to set the font size to 0.3 inches, call `'FontUnits','inches','FontSize',0.3` in the argument list.

'FontWeight' - Weight of text characters

`'normal'` (default) | `'bold'` | `'light'` | `'demi'`

Weight of text characters, specified as the comma-separated pair consisting of `'FontWeight'` and one of the following strings:

- `'normal'`
- `'bold'`
- `'light'`
- `'demi'`

MATLAB uses the `FontWeight` property to select a font from those available on your particular system. Generally, setting this property to `'bold'` or `'demi'` causes MATLAB to use a bold font.

Example: `'FontWeight','bold'`

'Interpreter' - Character interpretation

`'tex'` (default) | `'latex'` | `'none'`

Character interpretation, specified as the comma-separated pair consisting of `'Interpreter'` and one of the following strings.

Interpreter value	Result
'tex'	Supports a subset of plain TeX markup language. See the String property for a list of supported TeX instructions.
'latex'	Supports a basic subset of the LaTeX markup language.
'none'	Interprets all characters as literal characters.

Example: 'Interpreter','latex'

Output Arguments

h - Handle to text object used as title

Handle to the text object used as the title. This is a unique identifier, which you can use to query and modify the properties of the title.

Examples

Add Title to Current Figure

Create a figure and display a title in the current axes.

```
figure
plot((1:10).^2)
title('My Title')
```

You also can call title with a function that returns a string. For example, the date function returns a string containing today's date.

```
title(date)
```

MATLAB sets the output of date as the axes title.

Include Variable's Value in Title

Include the value of variable c in a title.

```
figure
plot((1:10).^2)
```

```
f = 70;
c = (f-32)/1.8;
title(['Temperature is ',num2str(c),' C'])
```

Create Multicolored Title Using TeX Markup

In a TeX string, use the color modifier `\color` to change the color of characters following it from the previous color.

```
figure
plot((1:10).^2)
title(['\fontsize{16}black {\color{magenta}magenta '...
'\color[rgb]{0 .5 .5}teal \color{red}red} black again'])
```

Create Colored Title Using Name,Value Pair Argument

Include the value of variable `n` in a title, and set the color of the title.

Use the `Name,Value` pair `'Color','y'` to set the color of the title to yellow.

```
figure
plot((1:10).^2)
title('Case number # 3','Color', 'y')
```

Include Greek Symbols in Title

Use a TeX string to include Greek symbols in a title.

```
t = (0:0.01:0.2);
y = exp(-25*t);
figure
plot(t,y)
title('y = \ite^{\lambda t}','Color','b')
```

The text object `String` property lists the available symbols. The `'Interpreter'` property must be `'tex'` (the default).

Include Superscript or Subscript Character in Title

```
figure
```

```
plot((1:10).^2)
title('\alpha^2 and X_1')
```

The superscript character "^" and the subscript character "_" modify the character or substring defined in braces immediately following.

Create Multiline Title

Create a multiline title using a multiline cell array.

```
figure
plot((1:10).^2)
title({'First line';'Second line'})
```

Display Text As Typed

Set the `Interpreter` property as `'none'` so that the string `X_1` is displayed in the figure as typed, without making `1` a subscript of `X`.

```
figure
plot((1:10).^2)
title('X_1','Interpreter','none')
```

MATLAB displays the string `X_1` in the title of the figure.

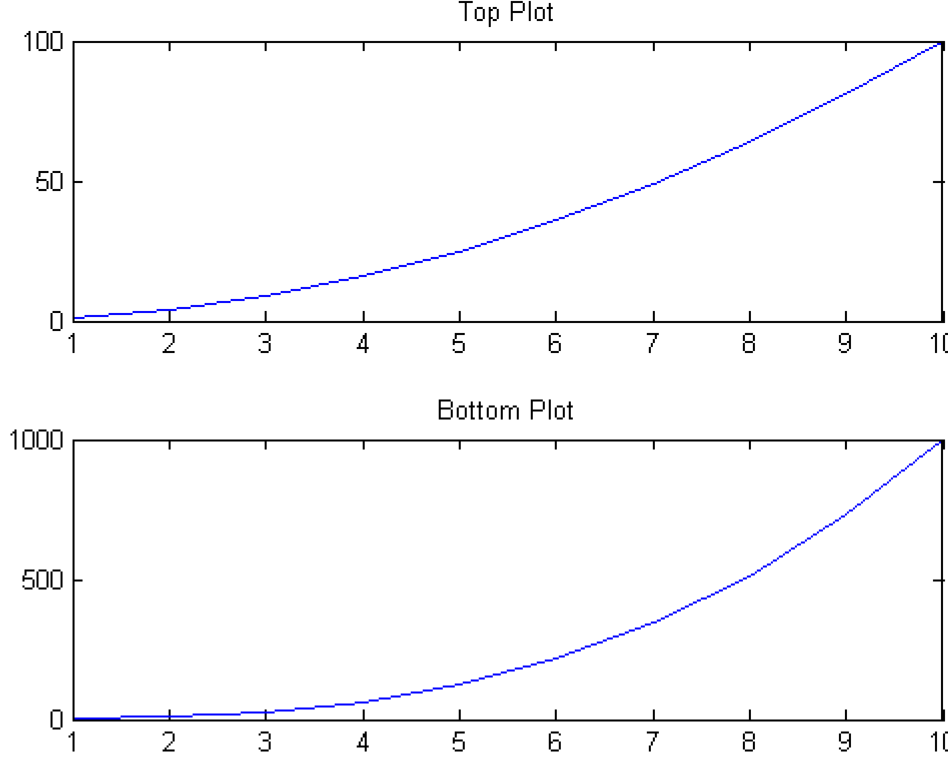
Add Title to Specific Axes

Create two subplots and return the handles to the axes objects, `s(1)` and `s(2)`.

```
figure
s(1) = subplot(2,1,1);
plot((1:10).^2)
s(2) = subplot(2,1,2);
plot((1:10).^3)
```

Add a title to each subplot by referring to its axes handle, `s(1)`, or `s(2)`.

```
title(s(1),'Top Plot')
title(s(2),'Bottom Plot')
```



Add Title and Return Object Handle

Add a title to a plot and return the handle to the text object used as the title.

```
figure
plot((1:10).^2)
h = title('My Title');
```

MATLAB returns the object handle in the output variable, `h`.

Set the color of the title to red, using the object handle.

```
set(h,'Color','red')
```

See Also

`gtext` | `int2str` | `num2str` | `text` | `xlabel` | `ylabel` | `zlabel` | `gca`

Concepts

- Adding Titles to Graphs

Text Properties

todatenum

Purpose

Convert CDF epoch object to MATLAB datenum

Syntax

```
n = todatenum(obj)
```

Description

`n = todatenum(obj)` converts the CDF epoch object `ep_obj` into a MATLAB serial date number. Note that a CDF epoch is the number of milliseconds since `01-Jan-0000` whereas a MATLAB datenum is the number of days since 00`-Jan-0000`.

Examples

Construct a CDF epoch object from a date string, and then convert the object back into a MATLAB date string:

```
dstr = datestr(today)
dstr =
   08-Oct-2003

obj = cdfepoch(dstr)
obj =
     cdfepoch object:
     08-Oct-2003 00:00:00

dstr2 = datestr(todatenum(obj))
dstr2 =
   08-Oct-2003
```

See Also

`cdfepoch` | `cdfinfo` | `cdfread` | `datenum`

toeplitz

Purpose

Toeplitz matrix

Syntax

```
T = toeplitz(c,r)
T = toeplitz(r)
```

Description

A *Toeplitz* matrix is defined by one row and one column. A *symmetric Toeplitz* matrix is defined by just one row. `toeplitz` generates Toeplitz matrices given just the row or row and column description.

`T = toeplitz(c,r)` returns a nonsymmetric Toeplitz matrix `T` having `c` as its first column and `r` as its first row. If the first elements of `c` and `r` are different, a message is printed and the column element is used.

For a real vector `r`, `T = toeplitz(r)` returns the symmetric Toeplitz matrix formed from vector `r`, where `r` defines the first row of the matrix. For a complex vector `r` with a real first element, `T = toeplitz(r)` returns the Hermitian Toeplitz matrix formed from `r`, where `r` defines the first row of the matrix and `r'` defines the first column. When the first element of `r` is not real, the resulting matrix is Hermitian off the main diagonal, i.e., $T_{ij} = \text{conj}(T_{ji})$ for $i \neq j$.

Examples

A Toeplitz matrix with diagonal disagreement is

```
c = [1  2  3  4  5];
r = [1.5  2.5  3.5  4.5  5.5];
toeplitz(c,r)
Column wins diagonal conflict:
ans =
    1.000    2.500    3.500    4.500    5.500
    2.000    1.000    2.500    3.500    4.500
    3.000    2.000    1.000    2.500    3.500
    4.000    3.000    2.000    1.000    2.500
    5.000    4.000    3.000    2.000    1.000
```

See Also

`hankel` | `kron`

Purpose

Root folder for specified toolbox

Syntax

```
toolboxdir('tbxFolderName')
s = toolboxdir('tbxFolderName')
s = toolboxdir tbxFolderName
```

Description

`toolboxdir('tbxFolderName')` returns a string that is the absolute path to the specified toolbox, `tbxFolderName`, where `tbxFolderName` is the folder name for the toolbox.

`s = toolboxdir('tbxFolderName')` returns the absolute path to the specified toolbox to the output argument, `s`.

`s = toolboxdir tbxFolderName` is the command form of the syntax.

Tips

`toolboxdir` is particularly useful for MATLAB Compiler software. The base folder of all toolboxes installed with MATLAB software is:

matlabroot/**toolbox**/tbxFolderName

However, in deployed mode, the base folders of the toolboxes are different. `toolboxdir` returns the correct root folder, whether running from MATLAB or from an application deployed with the MATLAB Compiler software.

To determine the folder name for a given toolbox, run the following code, substituting the name of a product function for *toolbxfcn*:

```
n = 'toolbxfcn';
pat = '(?<=^.+[\\/]toolbox[\\/])[^\\/]+';
regexp(which(n), pat, 'match', 'once')
```

For example, to determine the product name for Control System Toolbox set `n` to the name of a function unique to Control System Toolbox, such as `dss`:

```
n = 'dss'
pat = '(?<=^.+[\\/]toolbox[\\/])[^\\/]+'
regexp(which(n), pat, 'match', 'once')
```

`control`

Examples

Obtain the path for the Control System Toolbox software:

```
s = toolboxdir('control')
```

MATLAB returns:

```
s = C:\Program Files\MATLAB\R2012a\toolbox\control
```

See Also

`fullfile` | `matlabroot` | `path`

Purpose

Sum of diagonal elements

Syntax

```
b = trace(A)
```

Description

`b = trace(A)` is the sum of the diagonal elements of the matrix `A`.

Algorithms

```
t = sum(diag(A));
```

See Also

`det` | `eig`

transpose

Purpose Transpose

Syntax

```
b = a.'
b = transpose(a)
```

Description

`b = a.'` computes the non-conjugate transpose of matrix `a` and returns the result in `b`.

`b = transpose(a)` is called for the syntax `a.'` when `a` is an object.

See Also

`ctranspose` | `permute`

Purpose

Trapezoidal numerical integration

Syntax

```
Z = trapz(Y)
Z = trapz(X,Y)
Z = trapz(...,dim)
```

Description

`Z = trapz(Y)` computes an approximation of the integral of `Y` via the trapezoidal method (with unit spacing). To compute the integral for spacing other than one, multiply `Z` by the spacing increment. Input `Y` can be complex.

If `Y` is a vector, `trapz(Y)` is the integral of `Y`.

If `Y` is a matrix,`trapz(Y)` is a row vector with the integral over each column.

If `Y` is a multidimensional array, `trapz(Y)` works across the first nonsingleton dimension.

`Z = trapz(X,Y)` computes the integral of `Y` with respect to `X` using trapezoidal integration. Inputs `X` and `Y` can be complex.

If `X` is a column vector and `Y` an array whose first nonsingleton dimension is `length(X)`, `trapz(X,Y)` operates across this dimension.

`Z = trapz(...,dim)` integrates across the dimension of `Y` specified by scalar `dim`. The length of `X`, if given, must be the same as `size(Y,dim)`.

Examples

Example 1

The exact value of $\int_0^{\pi} \sin(x)\, dx$ is 2.
To approximate this numerically on a uniformly spaced grid, use

```
X = 0:pi/100:pi;
Y = sin(X);
```

Then both

```
Z = trapz(X,Y)
```

and

```
Z = pi/100*trapz(Y)
```

produce

```
Z =
    1.9998
```

Example 2

A nonuniformly spaced example is generated by

```
X = sort(rand(1,101)*pi);
Y = sin(X);
Z = trapz(X,Y);
```

The result is not as accurate as the uniformly spaced grid. One random sample produced

```
Z =
    1.9984
```

Example 3

This example uses two complex inputs:

```
z = exp(1i*pi*(0:100)/100);

trapz(z, 1./z)
ans =
   0.0000 + 3.1411i
```

See Also

cumsum | cumtrapz | integral | integral2 | integral3

Purpose

Lay out tree or forest

Syntax

```
[x,y] = treelayout(parent,post)
[x,y,h,s] = treelayout(parent,post)
```

Description

`[x,y] = treelayout(parent,post)` lays out a tree or a forest. `parent` is the vector of parent pointers, with `0` for a root. `post` is an optional postorder permutation on the tree nodes. If you omit `post`, `treelayout` computes it. `x` and `y` are vectors of coordinates in the unit square at which to lay out the nodes of the tree to make a nice picture.

`[x,y,h,s] = treelayout(parent,post)` also returns the height of the tree `h` and the number of vertices `s` in the top-level separator.

See Also

`etree` | `treeplot` | `etreeplot` | `symbfact`

treeplot

Purpose

Plot picture of tree

Syntax

```
treeplot(p)
treeplot(p,nodeSpec,edgeSpec)
```

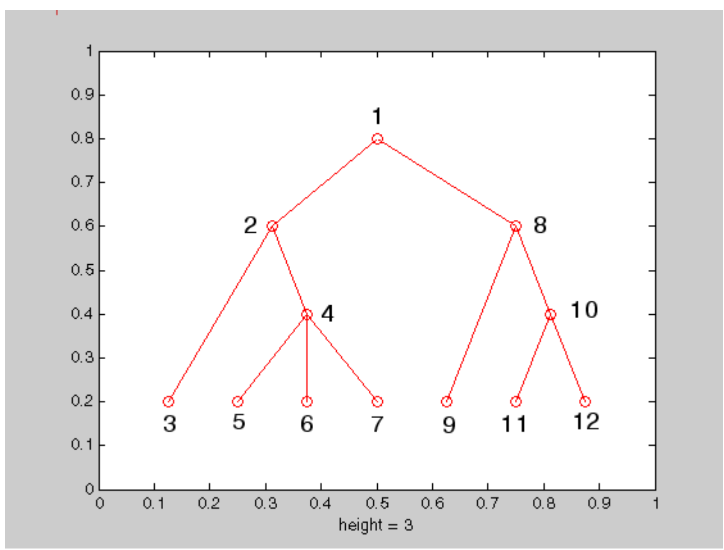
Description

`treeplot(p)` plots a picture of a tree given a vector of parent pointers, with `p(i)` = 0 for a root.

`treeplot(p,nodeSpec,edgeSpec)` allows optional parameters `nodeSpec` and `edgeSpec` to set the node or edge color, marker, and linestyle. Use `''` to omit one or both.

Examples

To plot a tree with 12 nodes, call `treeplot` with a 12-element input vector. The index of each element in the vector is shown adjacent to each node in the figure below. (These indices are shown only for the point of illustrating the example; they are not part of the `treeplot` output.)



To generate this plot, set the value of each element in the `nodes` vector to the index of its parent, (setting the parent of the root node to zero).

The node marked `1` in the figure is represented by `nodes(1)` in the input vector, and because this is the root node which has a parent of zero, you set its value to zero:

```
nodes(1) = 0;     % Root node
```

`nodes(2)` and `nodes(8)` are children of `nodes(1)`, so set these elements of the input vector to `1`:

```
nodes(2) = 1;     nodes(8) = 1;
```

`nodes(5:7)` are children of `nodes(4)`, so set these elements to `4`:

```
nodes(5) = 4;     nodes(6) = 4;     nodes(7) = 4;
```

Continue in this manner until each element of the vector identifies its parent. For the plot shown above, the `nodes` vector now looks like this:

```
nodes = [0 1 2 2 4 4 4 1 8 8 10 10];
```

Now call `treeplot` to generate the plot:

```
treeplot(nodes)
```

See Also

`etree` | `etreeplot` | `treelayout`

Purpose Lower triangular part of matrix

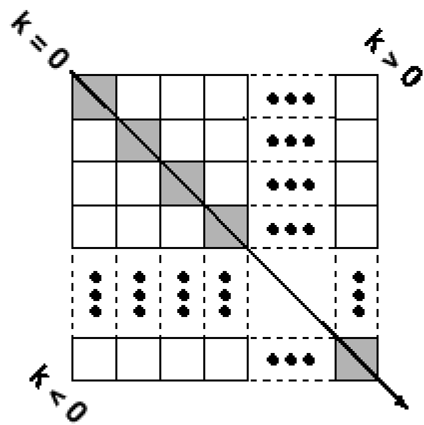
Syntax

```
L = tril(X)
L = tril(X,k)
```

Description

`L = tril(X)` returns the lower triangular part of `X`.

`L = tril(X,k)` returns the elements on and below the kth diagonal of `X`. `k = 0` is the main diagonal, `k > 0` is above the main diagonal, and `k < 0` is below the main diagonal.



Examples

```
tril(ones(4,4),-1)

ans =

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

See Also

`diag` | `triu`

Purpose

Triangular mesh plot

Syntax

```
trimesh(Tri,X,Y,Z,C)
trimesh(Tri,X,Y,Z)
trimesh(Tri,X,Y)
trimesh(TR)
trimesh(...'PropertyName',PropertyValue...)
h = trimesh(...)
```

Description

`trimesh(Tri,X,Y,Z,C)` displays triangles defined in the *m*-by-3 face matrix `Tri` as a mesh. Each row of `Tri` defines a single triangular face by indexing into the vectors or matrices that contain the `X`, `Y`, and `Z` vertices. The edge color is defined by the vector `C`.

`trimesh(Tri,X,Y,Z)` uses `C = Z` so color is proportional to surface height.

`trimesh(Tri,X,Y)` displays the triangles in a 2-D plot.

`trimesh(TR)` displays the triangles in a triangulation representation.

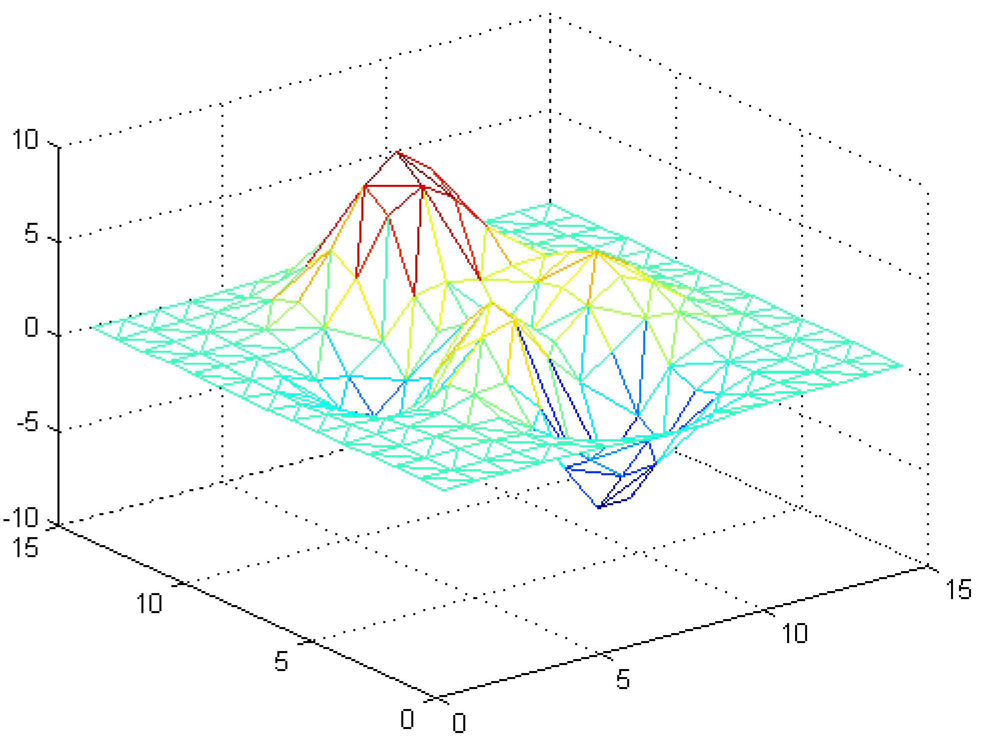
`trimesh(...'PropertyName',PropertyValue...)` specifies additional patch property names and values for the patch graphics object created by the function.

`h = trimesh(...)` returns a handle to the displayed triangles.

Examples

Create vertex vectors and a face matrix, then create a triangular mesh plot.

```
[x,y]=meshgrid(1:15,1:15);
tri = delaunay(x,y);
z = peaks(15);
trimesh(tri,x,y,z)
```



If the surface is already a triangulation representation, then you can pass the triangulation to `trimesh`:

```
tr = triangulation(tri,x(:),y(:),z(:));
trimesh(tr)
```

See Also

`patch` | `trisurf` | `delaunay` | `delaunayTriangulation` | `triangulation`

Purpose

Numerically evaluate triple integral

Compatibility

`triplequad` will be removed in a future release. Use `integral3` instead.

Syntax

```
q = triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax)
q = triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol)
q = triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol,method)
```

Description

`q = triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax)` evaluates the triple integral `fun(x,y,z)` over the three dimensional rectangular region `xmin <= x <= xmax`, `ymin <= y <= ymax`, `zmin <= z <= zmax`. The first input, `fun`, is a function handle. `fun(x,y,z)` must accept a vector `x` and scalars `y` and `z`, and return a vector of values of the integrand.

"Parameterizing Functions" explains how to provide additional parameters to the function `fun`, if necessary.

`q = triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol)` uses a tolerance `tol` instead of the default, which is `1.0e-6`.

`q = triplequad(fun,xmin,xmax,ymin,ymax,zmin,zmax,tol,method)` uses the quadrature function specified as `method`, instead of the default `quad`. Valid values for `method` are `@quadl` or the function handle of a user-defined quadrature method that has the same calling sequence as `quad` and `quadl`.

Examples

Pass function handle `@integrnd` to `triplequad`:P

```
Q = triplequad(@integrnd,0,pi,0,1,-1,1);
```

where the file `integrnd.m` is

```
function f = integrnd(x,y,z)
f = y*sin(x)+z*cos(x);
```

Pass anonymous function handle `F` to `triplequad`:

```
F = @(x,y,z)y*sin(x)+z*cos(x);
Q = triplequad(F,0,pi,0,1,-1,1);
```

This example integrates `y*sin(x)+z*cos(x)` over the region `0 <= x <= pi`, `0 <= y <= 1`, `-1 <= z <= 1`. Note that the integrand can be evaluated with a vector `x` and scalars `y` and `z`.

See Also

`dblquad` | `quad2d` | `quad` | `quadgk` | `quadl` | `function_handle` | `integral` | `integral2` | `integral3`

How To

- "Anonymous Functions"

Purpose

2-D triangular plot

Syntax

```
triplot(TRI,x,y)
triplot(TRI,x,y,color)
triplot(TR)
h = triplot(...)
triplot(...,'param','value','param','value'...)
```

Description

`triplot(TRI,x,y)` displays the triangles defined in the `m`-by-3 matrix `TRI`. A row of `TRI` contains indices into the vectors `x` and `y` that define a single triangle. The default line color is blue.

`triplot(TRI,x,y,color)` uses the string `color` as the line color. `color` can also be a line specification. See `ColorSpec` for a list of valid color strings. See `LineSpec` for information about line specifications.

`triplot(TR)` displays the triangles in a triangulation representation.

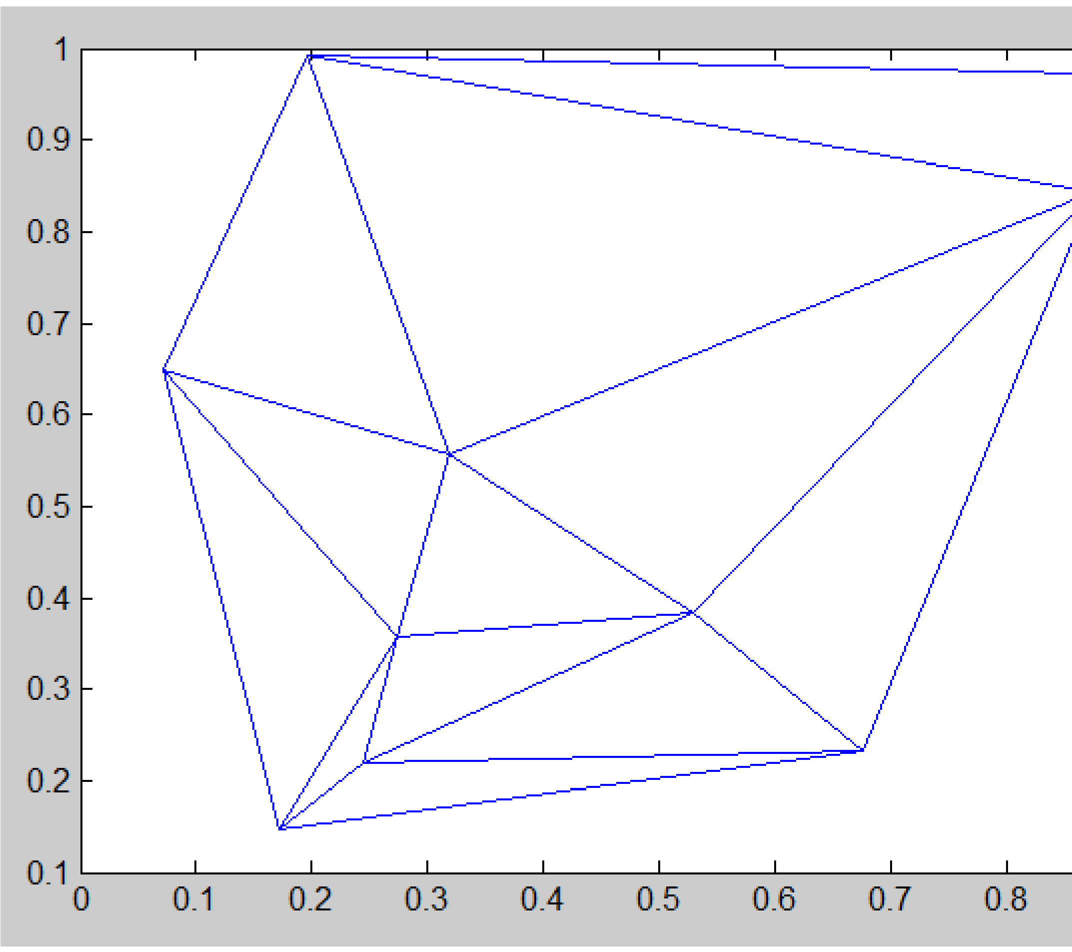
`h = triplot(...)` returns a vector of handles to the displayed triangles.

`triplot(...,'param','value','param','value'...)` allows additional line property name/property value pairs to be used when creating the plot. See `Line Properties` for information about the available properties.

Examples

Plot a Delaunay triangulation for 10 randomly generated points.

```
P = gallery('uniformdata',10,2,2);
DT = delaunayTriangulation(P);
triplot(DT)
```



See Also

`delaunayTriangulation` | `triangulation` | `delaunay` | `trimesh` | `trisurf`

Purpose

(Will be removed) Triangulation representation

Note `TriRep` will be removed in a future release. Use `triangulation` instead.

Description

`TriRep` provides topological and geometric queries for triangulations in 2-D and 3-D space. For example, for triangular meshes you can query triangles attached to a vertex, triangles that share an edge, neighbor information, circumcenters, or other features. You can create a `TriRep` directly using existing triangulation data. Alternatively, you can create a Delaunay triangulation, via `DelaunayTri`, which provides access to the `TriRep` functionality.

Construction

TriRep	(Will be removed) Triangulation representation

Methods

baryToCart	(Will be removed) Convert point coordinates from barycentric to Cartesian
cartToBary	(Will be removed) Convert point coordinates from cartesian to barycentric
circumcenters	(Will be removed) Circumcenters of specified simplices
edgeAttachments	(Will be removed) Simplices attached to specified edges
edges	(Will be removed) Triangulation edges

faceNormals	(Will be removed) Unit normals to specified triangles
featureEdges	(Will be removed) Sharp edges of surface triangulation
freeBoundary	(Will be removed) Facets referenced by only one simplex
incenters	(Will be removed) Incenters of specified simplices
isEdge	(Will be removed) Test if vertices are joined by edge
neighbors	(Will be removed) Simplex neighbor information
size	(Will be removed) Size of triangulation matrix
vertexAttachments	(Will be removed) Return simplices attached to specified vertices

Properties

X	Coordinates of the points in the triangulation
Triangulation	Triangulation data structure

Copy Semantics

Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

Indexing

`TriRep` objects support indexing into the triangulation using parentheses (). The syntax is the same as for arrays.

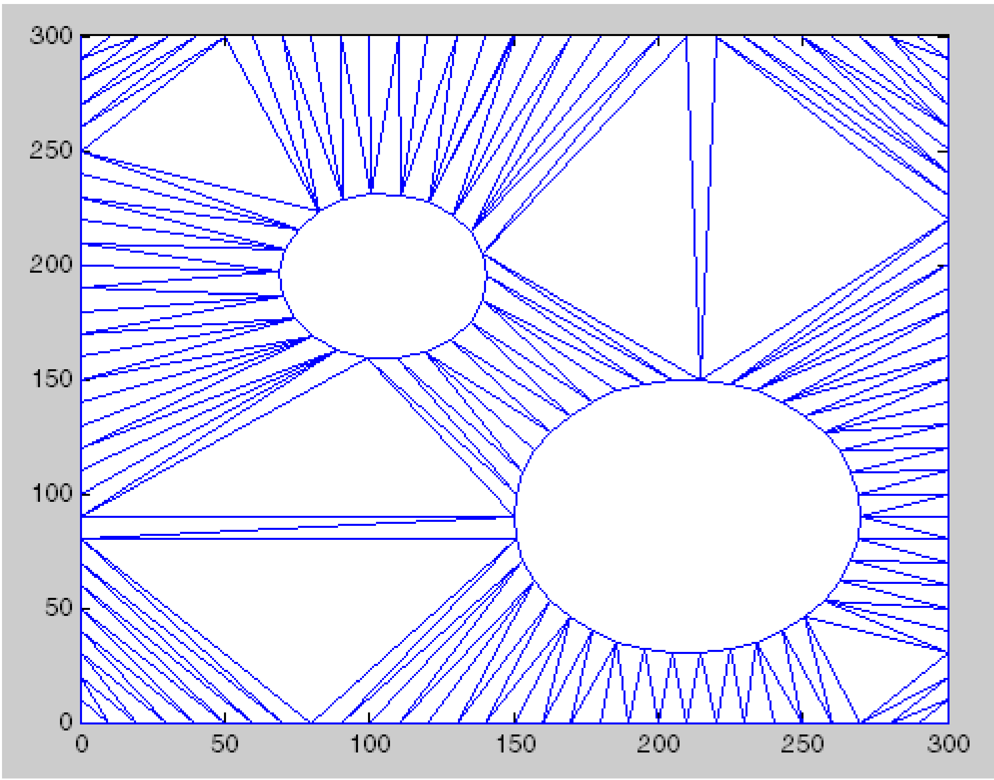
Examples

Load a 2-D triangulation and use the `TriRep` constructor to build an array of the free boundary edges:

```
load trimesh2d
```

This loads triangulation `tri` and vertex coordinates `x`, `y`:

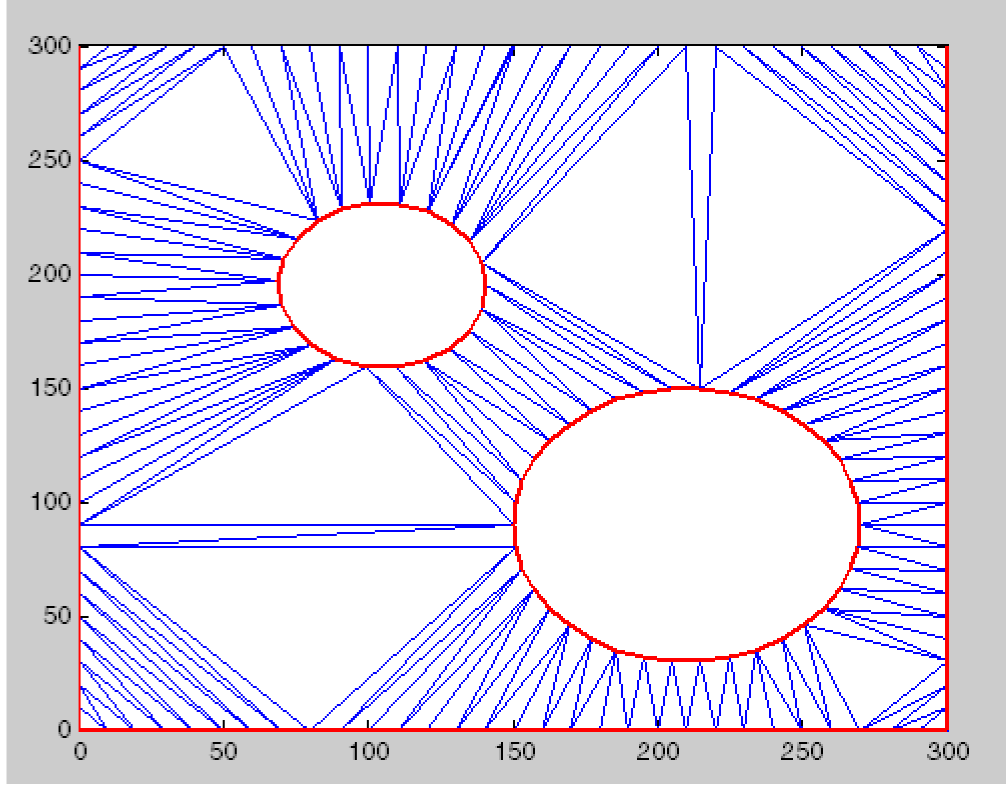
```
trep = TriRep(tri, x,y);
fe = freeBoundary(trep)';
triplot(trep);
```



You can add the free edges `fe` to the plot:

```
hold on;
plot(x(fe), y(fe), 'r','LineWidth',2);
hold off;
```

```
axis([-50 350 -50 350]);
axis equal;
```



See Also

delaunayTriangulation | triangulation | scatteredInterpolant

Purpose

(Will be removed) Triangulation representation

Note `TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
TR = TriRep(TRI, X, Y)
TR = TriRep(TRI, X, Y, Z)
TR = TriRep(TRI, X)
```

Description

`TR = TriRep(TRI, X, Y)` creates a 2-D triangulation representation from the triangulation matrix `TRI` and the vertex coordinates `(X, Y)`. `TRI` is an `m`-by-3 matrix that defines the triangulation in face-vertex format, where `m` is the number of triangles. Each row of `TRI` is a triangle defined by indices into the column vector of vertex coordinates `(X, Y)`.

`TR = TriRep(TRI, X, Y, Z)` creates a 3-D triangulation representation from the triangulation matrix `TRI` and the vertex coordinates `(X, Y, Z)`. `TRI` is an `m`-by-3 or `m`-by-4 matrix that defines the triangulation in simplex-vertex format, where where m is the number of simplices; triangles or tetrahedra in this case. Each row of `TRI` is a simplex defined by indices into the column vector of vertex coordinates `(X, Y, Z)`.

`TR = TriRep(TRI, X)` creates a triangulation representation from the triangulation matrix `TRI` and the vertex coordinates `X`. `TRI` is an `m`-by-`n` matrix that defines the triangulation in simplex-vertex format, where `m` is the number of simplices and `n` is the number of vertices per simplex. Each row of `TRI` is a simplex defined by indices into the array of vertex coordinates `X`. `X` is an `mpts`-by-`ndim` matrix where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside, where $2 \le$ `ndim` ≤ 3.

Examples

Load a 3-D tetrahedral triangulation compute the free boundary. First, load triangulation `tet` and vertex coordinates `X`.

```
load tetmesh
```

Create the triangulation representation and compute the free boundary.

```
trep = TriRep(tet, X);
[tri, Xb] = freeBoundary(trep);
```

See Also

scatteredInterpolant | delaunayTriangulation

Purpose

(Will be removed) Interpolate scattered data

Note `TriScatteredInterp` will be removed in a future release. Use `scatteredInterpolant` instead.

Description

`TriScatteredInterp` is used to perform interpolation on a scattered dataset that resides in 2-D or 3-D space. A scattered data set defined by locations `X` and corresponding values `V` can be interpolated using a Delaunay triangulation of `X`. This produces a surface of the form `V = F(X)`. The surface can be evaluated at any query location `QX`, using `QV = F(QX)`, where `QX` lies within the convex hull of `X`. The interpolant `F` always goes through the data points specified by the sample.

Definitions

The *Delaunay triangulation* of a set of points is a triangulation such that the unique circle circumscribed about each triangle contains no other points in the set. The *convex hull* of a set of points is the smallest convex set containing all points of the original set. These definitions extend naturally to higher dimensions.

Construction

TriScatteredInterp	(Will be removed) Interpolate scattered data

Properties

`X`	Defines locations of scattered data points in 2-D or 3-D space.
`V`	Defines value associated with each data point.

<table>
<tr><td rowspan="4">Method</td><td colspan="2">Defines method used to interpolate the data .</td></tr>
<tr><td>natural</td><td>Natural neighbor interpolation</td></tr>
<tr><td>linear</td><td>Linear interpolation (default)</td></tr>
<tr><td>nearest</td><td>Nearest neighbor interpolation</td></tr>
</table>

Copy Semantics

Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

Examples

Create a data set:

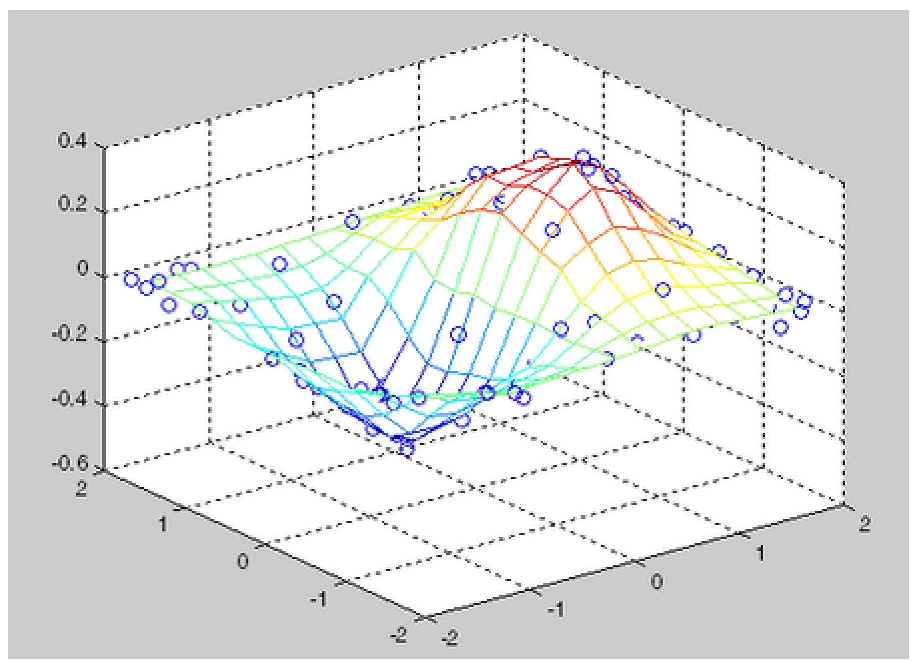
```
x = rand(100,1)*4-2;
y = rand(100,1)*4-2;
z = x.*exp(-x.^2-y.^2);
```

Construct the interpolant:

```
F = TriScatteredInterp(x,y,z);
```

Evaluate the interpolant at the locations (qx, qy). The corresponding value at these locations is qz:

```
ti = -2:.25:2;
[qx,qy] = meshgrid(ti,ti);
qz = F(qx,qy);
mesh(qx,qy,qz);
hold on;
plot3(x,y,z,'o');
```



See Also

delaunayTriangulation | interp1 | interp2 | interp3 | meshgrid

TriScatteredInterp

Purpose

(Will be removed) Interpolate scattered data

Note `TriScatteredInterp` will be removed in a future release. Use `scatteredInterpolant` instead.

Syntax

```
F = TriScatteredInterp()
F = TriScatteredInterp(X, V)
F = TriScatteredInterp(X, Y, V)
F = TriScatteredInterp(X, Y, Z, V)
F = TriScatteredInterp(DT, V)
F = TriScatteredInterp(..., method)
```

Description

`F = TriScatteredInterp()` creates an empty scattered data interpolant. This can subsequently be initialized with sample data points and values (`Xdata`, `Vdata`) via `F.X = Xdata` and `F.V = Vdata`.

`F = TriScatteredInterp(X, V)` creates an interpolant that fits a surface of the form `V = F(X)` to the scattered data in (`X`, `V`). `X` is a matrix of size `mpts`-by-`ndim`, where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside (`ndim` is 2 or 3). The column vector `V` defines the values at `X`, where the length of `V` equals `mpts`.

`F = TriScatteredInterp(X, Y, V)` and `F = TriScatteredInterp(X, Y, Z, V)` allow the data point locations to be specified in alternative column vector format when working in 2-D and 3-D.

`F = TriScatteredInterp(DT, V)` uses the specified `DelaunayTri` object `DT` as a basis for computing the interpolant. `DT` is a Delaunay triangulation of the scattered data locations, `DT.X`. The matrix `DT.X` is of size `mpts`-by-`ndim`, where `mpts` is the number of points and `ndim` is the dimension of the space where the points reside, `2 <= ndim <= 3`. `V` is a column vector that defines the values at `DT.X`, where the length of `V` equals `mpts`.

`F = TriScatteredInterp(..., method)` allows selection of the technique `method` used to interpolate the data.

Input Arguments

X	Matrix of size mpts-by-ndim, where mpts is the number of points and ndim is the dimension of the space where the points reside. Input may also be specified as column vectors (X, Y) or (X, Y, Z)	
V	Column vector that defines the values at X, where the length of V equals mpts.	
DT	Delaunay triangulation of the scattered data locations	
method	natural	Natural neighbor interpolation
	linear	Linear interpolation (default)
	nearest	Nearest-neighbor interpolation

Output Arguments

F	Creates an interpolant that fits a surface of the form V = F(X) to the scattered data.

Evaluation

To evaluate the interpolant, express the statement in Monge's form Vq = F(Xq), Vq = F(Xq,Yq), or Vq = F(Xq,Yq,Zq) where Vq is the value of the interpolant at the query location and Xq, Yq, and Zq are the vectors of point locations.

Definitions

The *Delaunay triangulation* of a set of points is a triangulation such that the unique circle circumscribed about each triangle contains no other points in the set.

Examples

Create a data set:

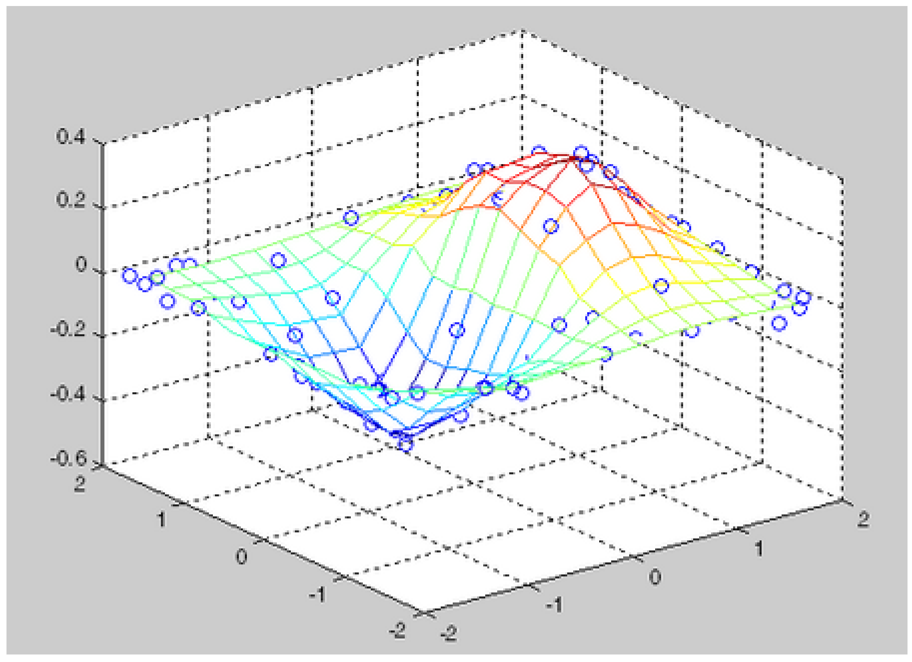
```
x = rand(100,1)*4-2;
y = rand(100,1)*4-2;
z = x.*exp(-x.^2-y.^2);
```

Construct the interpolant:

```
F = TriScatteredInterp(x,y,z);
```

Evaluate the interpolant at the locations (qx, qy). The corresponding value at these locations is qz .

```
ti = -2:.25:2;
[qx,qy] = meshgrid(ti,ti);
qz = F(qx,qy);
mesh(qx,qy,qz);
hold on;
plot3(x,y,z,'o');
```



See Also

delaunayTriangulation | interp1 | interp2 | interp3 | meshgrid

Purpose

Triangular surface plot

Syntax

```
trisurf(Tri,X,Y,Z,C)
trisurf(Tri,X,Y,Z)
trisurf(TR)
trisurf(...'PropertyName',PropertyValue...)
h = trisurf(...)
```

Description

`trisurf(Tri,X,Y,Z,C)` displays triangles defined in the *m*-by-3 face matrix `Tri` as a surface. Each row of `Tri` defines a single triangular face by indexing into the vectors or matrices that contain the `X`, `Y`, and `Z` vertices. The color is defined by the vector `C`.

`trisurf(Tri,X,Y,Z)` uses `C=Z` so color is proportional to surface height.

`trisurf(TR)` displays the triangles in a triangulation representation. It uses `C = TR.Points(:,3)` to make the the surface color is proportional to height.

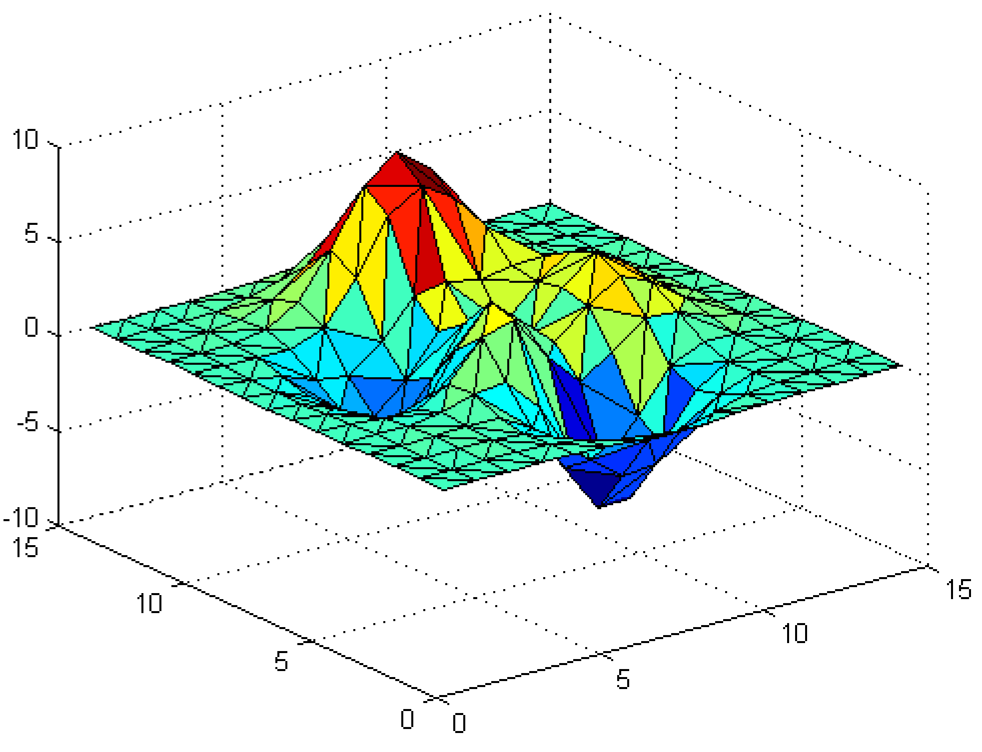
`trisurf(...'PropertyName',PropertyValue...)` specifies additional patch property names and values for the patch graphics object created by the function.

`h = trisurf(...)` returns a patch handle.

Examples

Create vertex vectors and a face matrix, then create a triangular surface plot.

```
[x,y] = meshgrid(1:15,1:15);
tri = delaunay(x,y);
z = peaks(15);
trisurf(tri,x,y,z)
```



If the surface is in the form of a triangulation representation, you can pass it to `trisurf` alone:

```
tr = triangulation(tri,x(:),y(:),z(:));
trisurf(tr)
```

See Also

`patch` | `surf` | `tetramesh` | `trimesh` | `triplot` | `delaunay` | `triangulation` | `delaunayTriangulation`

Purpose

Upper triangular part of matrix

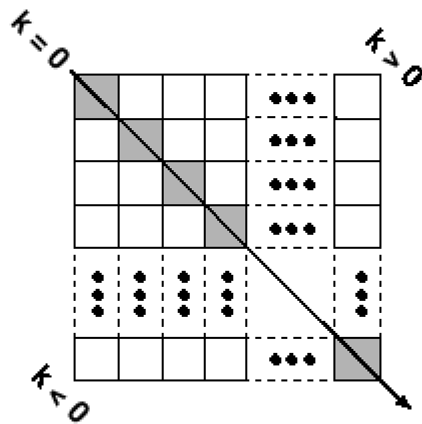
Syntax

```
U = triu(X)
U = triu(X,k)
```

Description

`U = triu(X)` returns the upper triangular part of `X`.

`U = triu(X,k)` returns the element on and above the kth diagonal of `X`. `k = 0` is the main diagonal, `k > 0` is above the main diagonal, and `k < 0` is below the main diagonal.



Examples

```
triu(ones(4,4),-1)

ans =

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

See Also

`diag` | `tril`

true

Purpose

Logical 1 (true)

Syntax

```
true
true(n)
true(m, n)
true(m, n, p, ...)
true(size(A))
true(...,'like',p)
```

Description

`true` is shorthand for `logical(1)`.

`true(n)` is an `n`-by-`n` matrix of logical ones.

`true(m, n)` or `true([m, n])` is an `m`-by-`n` matrix of logical ones.

`true(m, n, p, ...)` or `true([m n p ...])` is an `m`-by-`n`-by-`p`-by-`...` array of logical ones.

Note The size inputs `m`, `n`, `p`, `...` should be nonnegative integers. Negative integers are treated as 0.

`true(size(A))` is an array of logical ones that is the same size as array `A`.

`true(...,'like',p)` is an array of logical ones of the same data type and sparsity as the logical array `p`.

Tips

`true(n)` is much faster and more memory efficient than `logical(ones(n))`.

See Also

`false` | `logical`

Purpose

Execute statements and catch resulting errors

Syntax

```
try
   statements
catch exception
   statements
end
```

Description

`try` and `catch` blocks allow you to override the default error behavior for a set of program statements. If any statement in a `try` block generates an error, program control goes immediately to the `catch` block, which contains your error handling statements.

`exception` is an optional `MException` object input to the `catch` block that allows you to identify the error.

Both `try` and `catch` blocks can contain nested `try/catch` statements.

Examples

Provide more information about a dimension mismatch error:

```
A = rand(3);
B = ones(5);

try
   C = [A; B];
catch err

   % Give more information for mismatch.
   if (strcmp(err.identifier,'MATLAB:catenate:dimensionMismatch'))

      msg = sprintf('%s', ...
           'Dimension mismatch occured: First argument has ', ...
            num2str(size(A,2)), ' columns while second has ', ...
            num2str(size(B,2)), ' columns.');
      error('MATLAB:myCode:dimensions', msg);

   % Display any other errors as usual.
   else
```

```
      rethrow(err);
   end

end  % end try/catch
```

Catch an error when reading an image file, and try adjusting the file extension to resolve the error (`.jpg` and `.jpeg` are similar, as are `.tif` and `.tiff`):

```
function imageData = readImage(filename)
try
   imageData = imread(filename);
catch exception

   % Is the error because MATLAB could not find the file?
   if ~exist(filename, 'file')

      % Check for common typos in the extension.
      [~, ~, extension] = fileparts(filename);
      switch extension
         case '.jpg'
            altFilename = strrep(filename, '.jpg', '.jpeg');
         case '.jpeg'
            altFilename = strrep(filename, '.jpeg', '.jpg');
         case '.tif'
            altFilename = strrep(filename, '.tif', '.tiff');
         case '.tiff'
            altFilename = strrep(filename, '.tiff', '.tif');
         otherwise
            rethrow(exception);
      end

      % Try again, with modifed filename.
      try
         imageData = imread(altFilename);
      catch exception2
```

```
            % Rethrow original error.
            rethrow(exception)
        end

    else
        rethrow(exception)
    end

end
```

See Also

error | assert | MException

tscollection

Purpose

Create `tscollection` object

Syntax

```
tsc = tscollection(TimeSeries)
tsc = tscollection(Time)
tsc = tscollection(...,'Parameter',Value,...)
```

Description

`tsc = tscollection(TimeSeries)` creates a `tscollection` object `tsc` with one or more `timeseries` objects already in the MATLAB workspace. The argument `TimeSeries` can be a

- Single `timeseries` object
- Cell array of `timeseries` objects

`tsc = tscollection(Time)` creates an empty `tscollection` object with the time vector `Time`. When time values are date strings, you must specify `Time` as a cell array of date strings.

`tsc = tscollection(...,'Parameter',Value,...)` creates a `tscollection` object with optional parameter-value pairs you enter after specifying either a time vector or a `timeseries` object. You can specify the following parameter:

- `Name` — String that specifies the name of this `tscollection` object

Tips

Definition: Time Series Collection

A time series collection object is a MATLAB variable that groups several time series with a common time vector. The time series that you include in the collection are called members of this collection.

Properties of Time Series Collection Objects

This table lists the properties of the `tscollection` object. You can specify the `Time`, `TimeSeries`, and `Name` properties as input arguments in the constructor.

Property	Description
Name	`tscollection` name as a string. This can differ from the `tscollection` name in the MATLAB workspace.
Time	When `TimeInfo.StartDate` is empty, values are measured relative to `0` . When `TimeInfo.StartDate` is defined, values represent date strings measured relative to the `StartDate`. The length of `Time` must be the same as the first or the last dimension of `Data` for each collection .
TimeInfo	Contains fields for contextual information about `Time`: • `Units` — Time units with any of the following values: `'weeks'`, `'days'`, `'hours'`, `'minutes'`, `'seconds'`, `'milliseconds'`, `'microseconds'`, `'nanoseconds'` • `Start` — Start time • `End` — End time (read only) • `Increment` — Interval between subsequent time values. `NaN` when times are not uniformly sampled. • `Length` — Length of the time vector (read only) • `Format` — String defining the date string display format. See `datestr`. • `StartDate` — Date string defining the reference date. See `setabstime (tscollection)`. • `UserData` — Any additional user-defined information

Examples

The following example shows how to create a `tscollection` object.

1 Import the sample data.

```
load count.dat
```

2 Create three `timeseries` objects to store each set of data:

```
count1 = timeseries(count(:,1),1:24,'name', 'ts1');
count2 = timeseries(count(:,2),1:24,'name', 'ts2');
count3 = timeseries(count(:,3),1:24,'name', 'ts3');
```

3 Create a `tscollection` object named `tsc` and add to it two out of three time series already in the MATLAB workspace, by using the following syntax:

```
tsc = tscollection({count1 count2},'name','tsc')
```

See Also

`addts` | `datestr` | `setabstime (tscollection)` | `timeseries`

Purpose

Construct event object for `timeseries` object

Syntax

```
e = tsdata.event(Name,Time)
e = tsdata.event(Name,Time,'Datenum')
```

Description

`e = tsdata.event(Name,Time)` creates an event object with the specified Name that occurs at the time Time. Time can either be a real value or a date string.

`e = tsdata.event(Name,Time,'Datenum')` uses `'Datenum'` to indicate that the `Time` value is a serial date number generated by the `datenum` function. The `Time` value is converted to a date string after the event is created.

Tips

You add events by using the `addevent` method.

Fields of the `tsdata.event` object include the following:

- `EventData` — MATLAB array that stores any user-defined information about the event
- `Name` — String that specifies the name of the event
- `Time` — Time value when this event occurs, specified as a real number
- `Units` — Time units
- `StartDate` — A reference date, specified in MATLAB `datestr` format. `StartDate` is empty when you have a numerical (non-date-string) time vector.

tsearchn

Purpose

N-D closest simplex search

Syntax

```
t = tsearchn(X,TRI,XI)
[t,P] = tsearchn(X,TRI,XI)
```

Description

`t = tsearchn(X,TRI,XI)` returns the indices `t` of the enclosing simplex of the Delaunay triangulation `TRI` for each point in `XI`. `X` is an `m`-by-`n` matrix, representing `m` points in N-dimensional space. `XI` is a `p`-by-`n` matrix, representing `p` points in N-dimensional space. `tsearchn` returns `NaN` for all points outside the convex hull of `X`. `tsearchn` requires a triangulation `TRI` of the points `X` obtained from `delaunayn`.

`[t,P] = tsearchn(X,TRI,XI)` also returns the barycentric coordinate `P` of `XI` in the simplex `TRI`. `P` is a `p`-by-`n+1` matrix. Each row of `P` is the barycentric coordinate of the corresponding point in `XI`. It is useful for interpolation.

See Also

`delaunayTriangulation`

Purpose

Open Time Series Tools GUI

Syntax

```
tstool
tstool(ts)
tstool(tsc)
```

Note `tstool(sldata)` and `tstool(ModelDataLogs,'replace')` have been removed. To view logged signal data, use the Simulink Simulation Data Inspector instead.

Description

`tstool` starts the Time Series Tools GUI without loading any data.

`tstool(ts)` starts the Time Series Tools GUI and loads the time-series object `ts` from the MATLAB workspace.

`tstool(tsc)` starts the Time Series Tools GUI and loads the time-series collection object `tsc` from the MATLAB workspace.

See Also

`timeseries` | `tscollection`

How To

- "View Logged Signal Data with the Simulation Data Inspector"

type

Purpose

Display contents of file

Syntax

```
type('filename')
type filename
```

Description

`type('filename')` displays the contents of the specified file in the MATLAB Command Window. Use the full path for `filename`, or use a MATLAB relative partial path.

If you do not specify a file extension and there is no `filename` file without an extension, the `type` function adds the `.m` extension by default. The `type` function checks the folders specified in the MATLAB search path, which makes it convenient for listing the contents of files on the screen. Use `type` with `more on` to see the listing one screen at a time.

`type filename` is the command form of the syntax.

Examples

`type('foo.bar')` lists the contents of the file `foo.bar`.

`type foo` lists the contents of the file `foo.` If `foo` does not exist, `type foo` lists the contents of the file `foo.m`.

See Also

`cd` | `dbtype` | `delete` | `dir` | `more` | `path` | `what` | `who`

Purpose

Convert data types without changing underlying data

Syntax

```
Y = typecast(X, type)
```

Description

`Y = typecast(X, type)` converts a numeric value in `X` to the data type specified by `type`. Input `X` must be a full, noncomplex, numeric scalar or vector. The `type` input is a string set to one of the following: `'uint8'`, `'int8'`, `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'uint64'`, `'int64'`, `'single'`, or `'double'`.

`typecast` is different from the MATLAB `cast` function in that it does not alter the input data. `typecast` always returns the same number of bytes in the output `Y` as were in the input `X`. For example, casting the 16-bit integer 1000 to `uint8` with `typecast` returns the full 16 bits in two 8-bit segments (3 and 232) thus keeping its original value (3*256 + 232 = 1000). The `cast` function, on the other hand, truncates the input value to 255.

The output of `typecast` can be formatted differently depending on what system you use it on. Some computer systems store data starting with its most significant byte (an ordering called *big-endian*), while others start with the least significant byte (called *little-endian*).

Note MATLAB issues an error if `X` contains fewer values than are needed to make an output value.

Examples

Example 1

This example converts between data types of the same size:

```
typecast(uint8(255), 'int8')
ans =
   -1

typecast(int16(-1), 'uint16')
ans =
```

```
65535
```

Example 2

Set X to a 1-by-3 vector of 32-bit integers, then cast it to an 8-bit integer type:

```
X = uint32([1 255 256])
X =
           1          255          256
```

Running this on a little-endian system produces the following results. Each 32-bit value is divided up into four 8-bit segments:

```
Y = typecast(X, 'uint8')
Y =
   1   0   0   0  255   0   0   0   0   1   0   0
```

The third element of `X`, 256, exceeds the 8 bits that it is being converted to in `Y`(9) and thus overflows to `Y`(10):

```
Y(9:12)
ans =
    0    1    0    0
```

Note that `length(Y)` is equal to `4.*length(X)`. Also note the difference between the output of `typecast` versus that of `cast`:

```
Z = cast(X, 'uint8')
Z =
    1  255  255
```

Example 3

This example casts a smaller data type (`uint8`) into a larger one (`uint16`). Displaying the numbers in hexadecimal format makes it easier to see just how the data is being rearranged:

```
format hex
X = uint8([44 55 66 77])
X =
```

```
   2c   37   42   4d
```

The first `typecast` is done on a big-endian system. The four 8-bit segments of the input data are combined to produce two 16-bit segments:

```
Y = typecast(X, 'uint16')
Y =
   2c37   424d
```

The second is done on a little-endian system. Note the difference in byte ordering:

```
Y = typecast(X, 'uint16')
Y =
   372c   4d42
```

You can format the little-endian output into big-endian (and vice versa) using the `swapbytes` function:

```
Y = swapbytes(typecast(X, 'uint16'))
Y =
   2c37   424d
```

Example 4

This example attempts to make a 32-bit value from a vector of three 8-bit values. MATLAB issues an error because there are an insufficient number of bytes in the input:

```
format hex
typecast(uint8([120 86 52]), 'uint32')

Error using typecast
Too few input values to make output type.
```

Repeat the example, but with a vector of four 8-bit values, and it returns the expected answer:

```
typecast(uint8([120 86 52 18]), 'uint32')
ans =
```

```
12345678
```

See Also

cast | class | swapbytes

Purpose

Create container object to exclusively manage radio buttons and toggle buttons

Syntax

```
uibuttongroup('PropertyName1',Value1,'PropertyName2',Value2,
   ...)
handle = uibuttongroup(...)
```

Description

A uibuttongroup groups components and manages exclusive selection behavior for radio buttons and toggle buttons that it contains. It can also contain other user interface controls, axes, uipanels, and uibuttongroups. It cannot contain ActiveX controls.

`uibuttongroup('PropertyName1',Value1,'PropertyName2',Value2,...)` creates a visible container component in the current figure window. This component manages exclusive selection behavior for uicontrols of style `radiobutton` and `togglebutton`.

`handle = uibuttongroup(...)` creates a uibuttongroup object and returns a handle to it in `handle`.

A `uibuttongroup` object can have `axes`, `uicontrol`, `uipanel`, and `uibuttongroup` objects as children. However, only uicontrols of style `radiobutton` and `togglebutton` are managed by the component.

When programming a button group, you do not code callbacks for the individual buttons; instead, use its `SelectionChangeFcn` callback to manage responses to selections. The following example illustrates how you use `uibuttongroup` event data to do this.

For the children of a `uibuttongroup` object, the `Position` property is interpreted relative to the button group. If you move the button group, the children automatically move with it and maintain their positions in the button group.

If you have a button group that contains a set of radio buttons and toggle buttons and you want:

- An immediate action to occur when a radio button or toggle button is selected, you must include the code to control the radio and toggle buttons in the button group's `SelectionChangeFcn` callback function,

not in the individual toggle button `Callback` functions. See the `SelectionChangeFcn` property and the example on this reference page for more information.

- Another component such as a push button to base its action on the selection, then that component's `Callback` callback can get the handle of the selected radio button or toggle button from the button group's `SelectedObject` property.

Use the `Parent` property to specify the parent as a figure, uipanel, or uibuttongroup. If you do not specify a parent, `uibuttongroup` adds the button group to the current figure. If no figure exists, one is created.

See the Uibuttongroup Properties reference page for more information.

After creating a uibuttongroup, you can set and query its property values using `set` and `get`. Run `get(handle)` to see a list of properties and their current values. Run `set(handle)` to see a list of object properties you can set and their legal values.

Tips

If you set the `Visible` property of a `uibuttongroup` object to `'off'`, any child objects it contains (buttons, button groups, etc.) become invisible along with the `uibuttongroup` panel itself. However, doing this does *not* affect the settings of the `Visible` property of any of its child objects, even though all of them remain invisible until the button group's visibility is set to `'on'`. `uipanel` components also behave in this manner.

Do not use the `CreateFcn` of a button group to create the buttons it contains. You can toggle buttons created in the button group `CreateFcn` on and off, but clicking them does not trigger the button group `SelectionChangeCallback` and the buttons do not display the expected mutually-exclusive behavior.

Examples

This example creates a uibuttongroup with three radio buttons. It manages the radio buttons with the `SelectionChangeFcn` callback, `selcbk`.

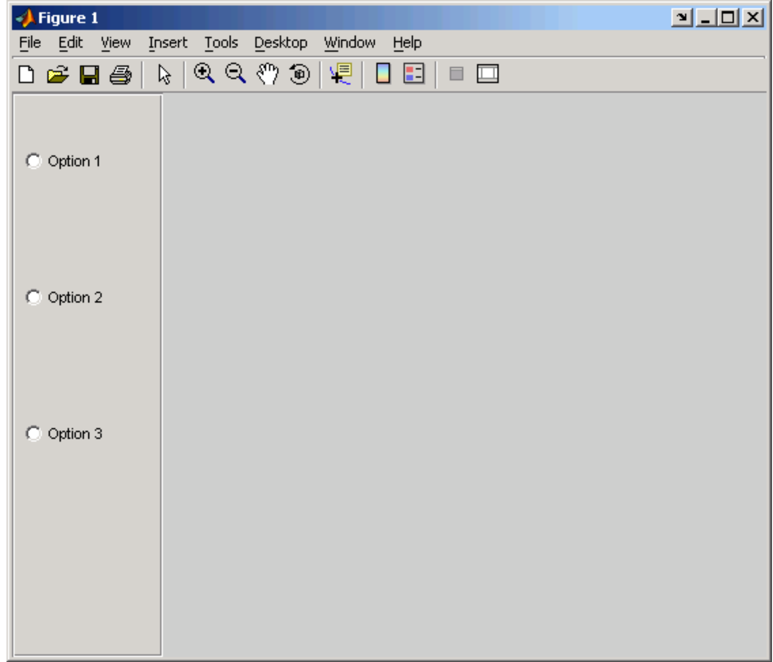
When you select a new radio button, `selcbk` displays the uibuttongroup handle on one line, the `EventName`, `OldValue`, and `NewValue` fields

of the event data structure on a second line, and the value of the `SelectedObject` property on a third line.

```
% Create the button group.
h = uibuttongroup('visible','off','Position',[0 0 .2 1]);
% Create three radio buttons in the button group.
u0 = uicontrol('Style','radiobutton','String','Option 1',...
    'pos',[10 350 100 30],'parent',h,'HandleVisibility','off');
u1 = uicontrol('Style','radiobutton','String','Option 2',...
    'pos',[10 250 100 30],'parent',h,'HandleVisibility','off');
u2 = uicontrol('Style','radiobutton','String','Option 3',...
    'pos',[10 150 100 30],'parent',h,'HandleVisibility','off');
% Initialize some button group properties.
set(h,'SelectionChangeFcn',@selcbk);
set(h,'SelectedObject',[]);  % No selection
set(h,'Visible','on');
```

For the `SelectionChangeFcn` callback, `selcbk`, the source and event data structure arguments are available only if `selcbk` is called using a function handle. See `SelectionChangeFcn` for more information.

```
function selcbk(source,eventdata)
disp(source);
disp([eventdata.EventName,'  ',...
     get(eventdata.OldValue,'String'),'  ', ...
     get(eventdata.NewValue,'String')]);
disp(get(get(source,'SelectedObject'),'String'));
```



If you click `Option 2` with no option selected, the `SelectionChangeFcn` callback, `selcbk`, displays:

```
    3.0011

SelectionChanged    Option 2
Option 2
```

If you then click `Option 1`, the `SelectionChangeFcn` callback, `selcbk`, displays:

```
    3.0011

SelectionChanged  Option 2  Option 1
Option 1
```

See Also

`uicontrol` | `uipanel`

Uibuttongroup Properties

Purpose

Describe button group properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

Uibuttongroup takes its default property values from uipanel. To set a uibuttongroup default property value, set the default for the corresponding `uipanel` property. Note that you can set no default values for the uibuttongroup `SelectedObject` and `SelectionChangeFcn` properties.

For more information about changing the default value of a property see "Setting Default Property Values". For an example, see the `CreateFcn` property.

Uibuttongroup Properties

This section describes all properties useful to `uibuttongroup` objects and lists valid values. Curly braces { } enclose default values.

Property Name	Description
`BackgroundColor`	Color of the button group background
`BeingDeleted`	This object is being deleted
`BorderType`	Type of border around the button group
`BorderWidth`	Width of the button group border in pixels
`BusyAction`	Interruption of other callback routines
`ButtonDownFcn`	Button-press callback routine
`Children`	All children of the button group

Property Name	Description
`Clipping`	Clipping of child axes, panels, and button groups to the button group. Does not affect child user interface controls (`uicontrol`)
`CreateFcn`	Callback routine executed during object creation
`DeleteFcn`	Callback routine executed during object deletion
`FontAngle`	Title font angle
`FontName`	Title font name
`FontSize`	Title font size
`FontUnits`	Title font units
`FontWeight`	Title font weight
`ForegroundColor`	Title font color and color of 2-D border line
`HandleVisibility`	Handle accessibility from command line and GUIs
`HighlightColor`	3-D frame highlight color
`Interruptible`	Callback routine interruption mode
`Parent`	`uibuttongroup` object's parent
`Position`	Button group position relative to parent figure, panel, or button group
`ResizeFcn`	User-specified resize routine
`Selected`	Whether object is selected
`SelectedObject`	Currently selected uicontrol of style `radiobutton` or `togglebutton`
`SelectionChangeFcn`	Callback routine executed when the selected radio button or toggle button changes
`SelectionHighlight`	Object highlighted when selected

Property Name	Description
ShadowColor	3-D frame shadow color
Tag	User-specified object identifier
Title	Title string
TitlePosition	Location of title string in relation to the button group
Type	Object class
UIContextMenu	Associate context menu with the button group
Units	Units used to interpret the position vector
UserData	User-specified data
Visible	Button group visibility **Note** Controls the visibility of a uibuttongroup and of its child axes, uibuttongroups. uipanels, and child uicontrols. Setting it does not change their `Visible` property.

`BackgroundColor`
: `ColorSpec`

Color of the uibuttongroup background. A three-element RGB vector or one of the MATLAB predefined names, specifying the background color. See the `ColorSpec` reference page for more information on specifying color.

`BeingDeleted`
: `on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called

(see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BorderType`

`none | {etchedin} | etchedout | beveledin | beveledout | line`

Border of the uibuttongroup area. Used to define the button group area graphically. Etched and beveled borders provide a 3-D look. Use the `HighlightColor` and `ShadowColor` properties to specify the border color of etched and beveled borders. A line border is 2-D. Use the `ForegroundColor` property to specify its color.

`BorderWidth`

integer

Width of the button group border. The width of the button group borders in pixels. The default border width is 1 pixel. 3-D borders wider than 3 may not appear correctly at the corners.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`
string or function handle

Button-press callback routine. A callback routine that executes when you press a mouse button while the pointer is in a 5-pixel wide border around the uibuttongroup. This is useful for implementing actions to interactively modify object properties, such as size and position, when they are clicked on (using the `selectmoveresize` function, for example).

If you define this routine as a string, the string can be a valid MATLAB expression or the name of a code file. The expression executes in the MATLAB workspace.

`Children`
vector of handles

Children of the uibuttongroup. A vector containing the handles of all children of the uibuttongroup. Although a uibuttongroup manages only uicontrols of style `radiobutton` and `togglebutton`, its children can be axes, uipanels, uibuttongroups, and other uicontrols. You can use this property to reorder the children.

`Clipping`
`{on} | off`

Clipping mode. By default, MATLAB clips a uibuttongroup's child axes, uipanels, and uibuttongroups to the uibuttongroup rectangle. If you set `Clipping` to `off`, the axis, uipanel, or

uibuttongroup is displayed outside the button group rectangle. This property does not affect child uicontrols which, by default, can display outside the button group rectangle.

`CreateFcn`

string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a `uibuttongroup` object. MATLAB sets all property values for the uibuttongroup before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the uibuttongroup being created.

Setting this property on an existing uibuttongroup object has no effect.

To define a default `CreateFcn` callback for all new uibuttongroups you must define the same default for all uipanels. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uibuttongroup`. For example, the code

```
set(0,'DefaultUipanelCreateFcn','set(gcbo,...
    ''FontName'',''arial'',''FontSize'',12)')
```

creates a default `CreateFcn` callback that runs whenever you create a new panel or button group. It sets the default font name and font size of the uipanel or uibuttongroup title.

To override this default and create a button group whose `FontName` and `FontSize` properties are set to different values, call `uibuttongroup` with code similar to

```
hpt = uibuttongroup(...,'CreateFcn','set(gcbo,...
''FontName'',''times'',''FontSize'',14)')
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uibuttongroup` call. In the example above, if instead of redefining the `CreateFcn` property for this uibuttongroup, you had explicitly set `FontSize` to `14`, the default `CreateFcn` callback would have set `FontSize` back to the system dependent default.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

See "Function Handle Callbacks" for information on how to use function handles to define a callback function.

`DeleteFcn`

string or function handle

Callback routine executed during object deletion. A callback routine that executes when you delete the `uibuttongroup` object (e.g., when you issue a delete command or clear the figure containing the uibuttongroup). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine. The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

`FontAngle`

`{normal} | italic | oblique`

Character slant used in the Title. MATLAB uses this property to select a font from those available on your particular system.

Setting this property to `italic` or `oblique` selects a slanted version of the font, when it is available on your system.

`FontName`
string

Font family used in the Title. The name of the font in which to display the `Title`. To display and print properly, this must be a font that your system supports. The default font is system dependent. To eliminate the need to hard code the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan), set `FontName` to the string `FixedWidth`. This string value is case insensitive.

```
set(uicontrol_handle,'FontName','FixedWidth')
```

This then uses the value of the root `FixedWidthFontName` property, which can be set to the appropriate value for a locale from `startup.m` in the end user's environment. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize`
integer

Title font size. A number specifying the size of the font in which to display the `Title`, in units determined by the `FontUnits` property. The default size is system dependent.

`FontUnits`
`inches | centimeters | normalized |`
`{points} |pixels`

Title font size units. `Normalized` units interpret `FontSize` as a fraction of the height of the uibuttongroup. When you resize the uibuttongroup, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

`FontWeight`
: `light | {normal} | demi | bold`

Weight of characters in the title. MATLAB uses this property to select a font from those available on your particular system. Setting this property to bold causes MATLAB to use a bold version of the font, when it is available on your system.

`ForegroundColor`
: `ColorSpec`

Color used for title font and 2-D border line. A three-element RGB vector or one of the MATLAB predefined names, specifying the font or line color. See the `ColorSpec` reference page for more information on specifying color.

`HandleVisibility`
: `{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

Note Uicontrols of style `radiobutton` and `togglebutton` that are managed by a uibuttongroup should not be accessed outside the button group. Set the `HandleVisibility` of such radio buttons and toggle buttons to `off` or `callback` to prevent inadvertent access.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

`HighlightColor`
`ColorSpec`

3-D frame highlight color. A three-element RGB vector or one of the MATLAB predefined names, specifying the highlight color. See the `ColorSpec` reference page for more information on specifying color.

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For user interface objects, the `Interruptible` property affects the callbacks for these properties only:

- `ButtonDownFcn`
- `KeyPressFcn`
- `KeyReleaseFcn`
- `WindowButtonDownFcn`
- `WindowButtonMotionFcn`
- `WindowButtonUpFcn`
- `WindowKeyPressFcn`
- `WindowKeyReleaseFcn`
- `WindowScrollWheelFcn`

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

The `BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting the `Interruptible` property to `on` (default), allows a callback from other user interface objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see "Control Callback Execution and Interruption".

Note If the interrupting callback is a `DeleteFcn`nor `CreateFcn` callback, or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. An object's `ButtonDownFcn` or `Callback` routine is processed according to the rules described previously in this section.

`Parent`
handle

Uibuttongroup parent. The handle of the uibuttongroup's parent figure, uipanel, or uibuttongroup. You can move a uibuttongroup object to another figure, uipanel, or uibuttongroup by setting this property to the handle of the new parent.

`Position`
position rectangle

Size and location of uibuttongroup relative to parent. The rectangle defined by this property specifies the size and location of the button group within the parent figure window, uipanel, or uibuttongroup. Specify `Position` as

```
[left bottom width height]
```

`left` and `bottom` are the distance from the lower-left corner of the parent object to the lower-left corner of the `uibuttongroup` object. `width` and `height` are the dimensions of the uibuttongroup rectangle, including the title. All measurements are in units specified by the `Units` property.

`ResizeFcn`
string or function handle

Resize callback routine. MATLAB executes this callback routine whenever a user resizes the uibuttongroup and the figure `Resize` property is set to `on`, or in GUIDE, the **Resize behavior** option is set to `Other`. You can query the uibuttongroup `Position` property to determine its new size and position. During execution of the callback routine, the handle to the figure being resized is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

You can use `ResizeFcn` to maintain a GUI layout that is not directly supported by the MATLAB `Position`/`Units` paradigm.

For example, consider a GUI layout that maintains an object at a constant height in pixels and attached to the top of the figure, but always matches the width of the figure. The following `ResizeFcn` accomplishes this; it keeps the uicontrol whose `Tag` is `'StatusBar'` 20 pixels high, as wide as the figure, and attached to the top of the figure. Note the use of the `Tag` property to retrieve the uicontrol handle, and the `gcbo` function to retrieve the figure handle. Also note the defensive programming regarding figure `Units`, which the callback requires to be in pixels in order to work

correctly, but which the callback also restores to their previous value afterwards.

```
u = findobj('Tag','StatusBar');
fig = gcbo;
old_units = get(fig,'Units');
set(fig,'Units','pixels');
figpos = get(fig,'Position');
upos = [0, figpos(4) - 20, figpos(3), 20];
set(u,'Position',upos);
set(fig,'Units',old_units);
```

You can change the figure `Position` from within the `ResizeFcn` callback; however, the `ResizeFcn` is not called again as a result.

Note that the `print` command can cause the `ResizeFcn` to be called if the `PaperPositionMode` property is set to `manual` and you have defined a resize function. If you do not want your resize function called by `print`, set the `PaperPositionMode` to `auto`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

`Selected`
`on | off` (read only)

Is object selected? This property indicates whether the button group is selected. When this property is `on`, MATLAB displays selection handles if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` function to set this property, allowing users to select the object with the mouse.

`SelectedObject`
scalar handle

Currently selected radio button or toggle button uicontrol in the managed group of components. Use this property to determine the currently selected component or to initialize selection of one of

the radio buttons or toggle buttons. By default, `SelectedObject` is set to the first uicontrol radio button or toggle button that is added. Set it to `[]` if you want no selection. Note that `SelectionChangeFcn` does not execute when this property is set by the user.

`SelectionChangeFcn`
string or function handle

Callback routine executed when the selected radio button or toggle button changes. If this routine is called as a function handle, `uibuttongroup` passes it two arguments. The first argument, `source`, is the handle of the uibuttongroup. The second argument, `eventdata`, is an event data structure that contains the fields shown in the following table.

Event Data Structure Field	Description
`EventName`	`'SelectionChanged'`
`OldValue`	Handle of the object selected before this event. `[]` if none was selected.
`NewValue`	Handle of the currently selected object.

If you have a button group that contains a set of radio buttons and/or toggle buttons and you want an immediate action to occur when a radio button or toggle button is selected, you must include the code to control the radio and toggle buttons in the button group's `SelectionChangeFcn` callback function, not in the individual toggle button `Callback` functions.

If you want another component such as a push button to base its action on the selection, then that component's `Callback` callback can get the handle of the selected radio button or toggle button from the button group's `SelectedObject` property.

Note For GUIDE GUIs, `hObject` contains the handle of the selected radio button or toggle button. See "Add Code for Components in Callbacks" for more information.

`SelectionHighlight`
`{on} | off`

Object highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`ShadowColor`
`ColorSpec`

3-D frame shadow color. `ShadowColor` is a three-element RGB vector or one of the MATLAB predefined names, specifying the shadow color. See the `ColorSpec` reference page for more information on specifying color.

`Tag`
string

User-specified object identifier. The `Tag` property provides a means to identify graphics objects with a user-specified label. You can define `Tag` as any string.

With the `findobj` function, you can locate an object with a given `Tag` property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified figures) that have the `Tag` value `'FormatTb'`.

```
h = findobj(figurehandles,'Tag','FormatTb')
```

Uibuttongroup Properties

`Title`
string

Title string. The text displayed in the button group title. You can position the title using the `TitlePosition` property.

If the string value is specified as a cell array of strings or padded string matrix, only the first string in the cell array or padded string matrix is displayed; the rest are ignored. Vertical slash ('`|`') characters are not interpreted as line breaks and instead show up in the text displayed in the uibuttongroup title.

Setting a property value to `default`, `remove`, or `factory` produces the effect described in "Defining Default Values". To set `Title` to one of these words, you must precede the word with the backslash character. For example,

```
hp = uibuttongroup(...,'Title','\Default');
```

`TitlePosition`
`{lefttop} | centertop | righttop |`
`leftbottom | centerbottom | rightbottom`

Location ofthe title. This property determines the location of the title string, in relation to the uibuttongroup.

`Type`
`string` (read-only)

Object class. This property identifies the kind of graphics object. For uibuttongroup objects, `Type` is always the string `'uipanel'`, because its default properties derive from uipanels.

`UIContextMenu`
handle

Associate a context menu with a uibuttongroup. Assign this property the handle of a `Uicontextmenu` object. MATLAB displays

the context menu whenever you right-click the `uibuttongroup`. Use the `uicontextmenu` function to create the context menu.

`Units`

`inches | centimeters | {normalized} |`
`points | pixels | characters`

Units of measurement. MATLAB uses these units to interpret the `Position` property. For the button group itself, units are measured from the lower-left corner of its parent figure window, panel, or button group. For children of the button group, they are measured from the lower-left corner of the button group.

- `Normalized` units map the lower-left corner of the button group or figure window to (0,0) and the upper-right corner to (1.0,1.0).
- `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).
- `Character` units are characters using the default system font; the width of one character is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is set to the default value.

`UserData`

matrix

User-specified data. Any data you want to associate with the `uibuttongroup` object. MATLAB does not use this data, but you can access it using `set` and `get`.

`Visible`

`{on} | off`

Uibuttongroup visibility. By default, a `uibuttongroup` object is visible. When set to `'off'`, the uibuttongroup is not visible, as are all child objects of the button group. When a button group is hidden in this manner, you can still query and set its properties.

Note The value of a uibuttongroup's `Visible` property determines whether its child components, such as axes, buttons, uipanels, and other uibuttongroups, are visible. However, changing the `Visible` property of a button group does *not* change the settings of the `Visible` property of its child components even though hiding the button group causes them to be hidden.

Purpose

Create context menu

Syntax

```
handle = uicontextmenu('PropertyName',PropertyValue,...)
```

Description

`handle = uicontextmenu('PropertyName',PropertyValue,...)` creates a context menu, which is a menu that appears when the user right-clicks on a graphics object. See Uicontextmenu Properties for more information.

In its initial state, a context menu has no menu items. You create menu items within the context menu using the `uimenu` function. Menu items appear in the order in which the `uimenu` statements appear. You then associate a context menu with an object by specifying the handle of the context menu as the value for its `UIContextMenu` property.

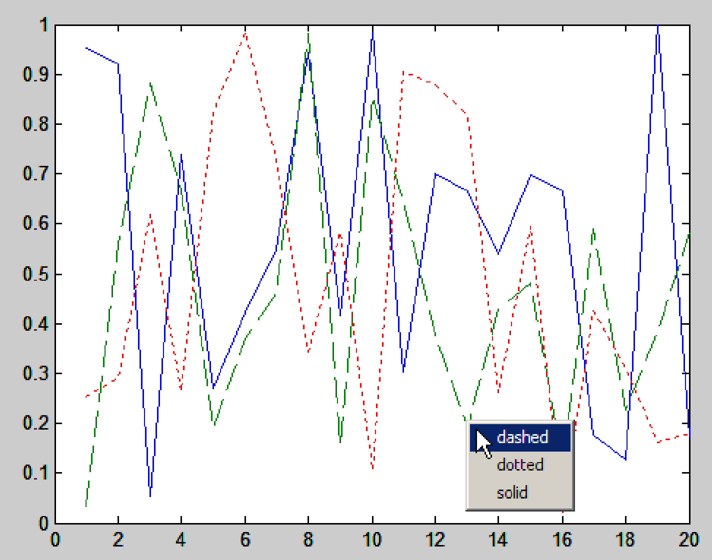
Examples

The following statements define a context menu associated with a line on a graph. The menu items enable you to change the line style.

```
% Create axes and save handle
hax = axes;
% Plot three lines
plot(rand(20,3));
% Define a context menu; it is not attached to anything
hcmenu = uicontextmenu;
% Define callbacks for context menu items that change linestyle
hcb1 = ['set(gco, ''LineStyle'', ''--'')'];
hcb2 = ['set(gco, ''LineStyle'', '':'')'];
hcb3 = ['set(gco, ''LineStyle'', ''-'')'];
% Define the context menu items and install their callbacks
item1 = uimenu(hcmenu, 'Label', 'dashed', 'Callback', hcb1);
item2 = uimenu(hcmenu, 'Label', 'dotted', 'Callback', hcb2);
item3 = uimenu(hcmenu, 'Label', 'solid',  'Callback', hcb3);
% Locate line objects
hlines = findall(hax,'Type','line');
% Attach the context menu to each line
for line = 1:length(hlines)
    set(hlines(line),'uicontextmenu',hcmenu)
```

```
end
```

When you right-click on any line (or, on a Macintosh computer with a one-button mouse, press the **Ctrl** key and click), the context menu appears, as shown in the following figure.



To make context menus available immediately, attach them to lines at the time they are plotted. Therefore, when creating a GUI that uses such context menus, place code like the preceding in the callbacks that perform plotting for the GUI.

A best practice is to use function handles for callbacks. Only define callbacks as strings for simple actions. For example, you can add check marks to menu items (using the `Checked` uimenu property) to indicate the current style for each line. To manage the check marks, define the menu item callbacks as function handles. Place the code for the functions in the GUI code file rather than placing callback strings in the figure.

Generally, you need to attach context menus to lines at the time they are plotted in order to be sure that the menus are immediately available. Therefore, code such as the above could be placed in or called from the callbacks that perform plotting for the GUI.

Tutorials

See "Context Menus" in the MATLAB Creating Graphical User Interfaces documentation.

See Also

`uibuttongroup` | `uicontrol` | `uimenu` | `uipanel`

Uicontextmenu Properties

Purpose

Describe context menu properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

For more information about changing the default value of a property see "Setting Default Property Values". For an example, see the `CreateFcn` property.

Uicontext-menu Properties

This section lists all properties useful to `uicontextmenu` objects along with valid values and descriptions of their use. Curly braces `{}` enclose default values.

Property	Purpose
`BeingDeleted`	This object is being deleted
`BusyAction`	Callback routine interruption
`Callback`	Control action
`Children`	The uimenus defined for the uicontextmenu
`CreateFcn`	Callback routine executed during object creation
`DeleteFcn`	Callback routine executed during object deletion
`HandleVisibility`	Whether handle is accessible from command line and GUIs
`Interruptible`	Callback routine interruption mode
`Parent`	Uicontextmenu object's parent

Property	Purpose
`Position`	Location of uicontextmenu when `Visible` is `on`
`Tag`	User-specified object identifier
`Type`	Class of graphics object
`UserData`	User-specified data
`Visible`	Uicontextmenu visibility

`BeingDeleted`

`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`Callback`
string

Control action. A routine that executes whenever you right-click an object for which a context menu is defined. The routine executes immediately before the context menu is posted. Define this routine as a string that is a valid MATLAB expression or the name of a code file. The expression executes in the MATLAB workspace.

`Children`
matrix

The uimenu items defined for the uicontextmenu.

`CreateFcn`
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a `uicontextmenu` object. MATLAB sets all property values for the uicontextmenu before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the uicontextmenu being created.

Setting this property on an existing uicontextmenu object has no effect.

You can define a default `CreateFcn` callback for all new uicontextmenus. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uicontextmenu`. For example, the code

```
set(0,'DefaultUicontextmenuCreateFcn','set(gcbo,...
    ''Visible'',''on'')')
```

creates a default `CreateFcn` callback that runs whenever you create a new context menu. It sets the default `Visible` property of a context menu.

To override this default and create a context menu whose `Visible` property is set to a different value, call `uicontextmenu` with code similar to

```
hpt = uicontextmenu(...,'CreateFcn','set(gcbo,...
''Visible'',''off'')')
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uicontextmenu` call. In the example above, if instead of redefining the `CreateFcn` property for this uicontextmenu, you had explicitly set `Visible` to `off`, the default `CreateFcn` callback would have set `Visible` back to the default, i.e., `on`.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`DeleteFcn`

string or function handle

Delete uicontextmenu callback routine. A callback routine that executes when you delete the uicontextmenu object (for example, when you issue a `delete` command or clear the figure containing the uicontextmenu). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define the callback function.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from

command-line users, while allowing callback routines to have complete access to object handles.

- Setting `HandleVisibility` to `off` makes handles invisible at all times. This might be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For user interface objects, the `Interruptible` property affects the callbacks for these properties only:

- `ButtonDownFcn`
- `KeyPressFcn`
- `KeyReleaseFcn`
- `WindowButtonDownFcn`
- `WindowButtonMotionFcn`
- `WindowButtonUpFcn`
- `WindowKeyPressFcn`
- `WindowKeyReleaseFcn`
- `WindowScrollWheelFcn`

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

The `BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting the `Interruptible` property to `on` (default), allows a callback from other user interface objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted.

For more information, see “Control Callback Execution and Interruption”.

Note If the interrupting callback is a `DeleteFcn`or `CreateFcn` callback, or a figure’s `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object’s `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. An object’s `ButtonDownFcn` or `Callback` routine is processed according to the rules described previously in this section.

`Parent`
handle

Uicontextmenu’s parent. The handle of the uicontextmenu’s parent object, which must be a figure.

`Position`
vector

Uicontextmenu’s position. A two-element vector that defines the location of a context menu posted by setting the `Visible` property value to `on`. Specify `Position` as

```
[x y]
```

where vector elements represent the horizontal and vertical distances in pixels from the bottom left corner of the figure window.

`Tag`
string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This

is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

`Type`
string

Class of graphics object. For uicontextmenu objects, `Type` is always the string `'uicontextmenu'`.

`UserData`
matrix

User-specified data. Any data you want to associate with the uicontextmenu object. MATLAB does not use this data, but you can access it using `set` and `get`.

`Visible`
`on | {off}`

Uicontextmenu visibility. The `Visible` property can be used in two ways:

- Its value indicates whether the context menu is currently posted. While the context menu is posted, the property value is `on`; when the context menu is not posted, its value is `off`.
- Its value can be set to `on` to force the posting of the context menu. Similarly, setting the value to `off` forces the context menu to be removed. When used in this way, the `Position` property determines the location of the posted context menu.

See Also

`uicontextmenu`

Purpose

Create user interface control object

Syntax

```
handle = uicontrol('Name',Value,...)
handle = uicontrol(parent,'Name',Value,...)
handle = uicontrol
uicontrol(uich)
```

Description

`handle = uicontrol('Name',Value,...)` creates a uicontrol and assigns the specified properties and values to it. It assigns the default values to any properties you do not specify. The default uicontrol style is `pushbutton`. The default parent is the current figure. See the Uicontrol Properties reference page for more information.

`handle = uicontrol(parent,'Name',Value,...)` creates a uicontrol in the object specified by the handle, `parent`. If you also specify a different value for the `Parent` property, the value of the `Parent` property takes precedence. `parent` can be the handle of a figure, uipanel, or uibuttongroup.

`handle = uicontrol` creates a push button in the current figure. The `uicontrol` function assigns all properties their default values.

`uicontrol(uich)` gives focus to the uicontrol specified by the handle, `uich`.

`uicontrol` creates a uicontrol graphics objects (user interface controls), which you use to implement graphical user interfaces.

Specifying the Uicontrol Style

When selected, most uicontrol objects perform a predefined action. To create a specific type of uicontrol, set the `Style` property as one of the strings that follow. You can specify a partial string if it is unique among all the styles. For example, instead of `'radiobutton'`, you can specify `'radio'`.

- `'checkbox'` – A check box generates an action when you select it. Use check boxes to provide a number of independent choices. To activate a check box, click the mouse button on the object. The check box updates its appearance when its state changes.

- `'edit'` – Editable text fields enable you to enter or modify text values. Use editable text when you want free text as input. To enable multiple lines of text, set `Max-Min>1`. Multiline edit boxes provide a vertical scroll bar for scrolling. The arrow keys also provide a way to scroll. Obtain the current text by getting the `String` property. The `String` property does not update as you type in an edit box. To execute the callback routine for an edit text control, type in the desired text and then do one of the following:
  - Click another component, the menu bar, or the background of the GUI.
  - For a single line editable text box, press **Enter**.
  - For a multiline editable text box, press **Ctl+Enter**.
- `'frame'`

  Note MathWorks recommends you use `uipanel` or `uibuttongroup` instead of frames.

  GUIDE continues to support frames in those GUIs that contain them, but the frame component does not appear in the GUIDE Layout Editor component palette.

- `'listbox'` – List boxes display a list of items, from which you can select one or more items. Unlike pop-up menus, list boxes do not expand when clicked. The `Min` and `Max` properties control the selection mode:
  - To enable multiple selection of items, set `Max-Min > 1`.
  - To enable selection of only one item at a time, set `Max-Min <= 1`

  The `Value` property stores the row indexes of currently selected list box items, and is a vector value when you select multiple items. After any mouse button up event that changes the `Value` property, MATLAB evaluates the list box's callback routine. To delay action

when multiple items can be selected, you can associate a "Done" push button with the list box. Use the callback for that button to evaluate the list box `Value` property.

List boxes with the `Enable` property set to `on` differentiate between single and double left clicks. MATLAB sets the figure `SelectionType` property to `normal` or `open` accordingly before evaluating the list box `Callback` property. For enabled list boxes, **Ctrl**-left click and **Shift**-left click also set the figure `SelectionType` property to `normal` or `open`, respectively indicating a single or double click.

- `'popupmenu'` – Pop-up menus (also known as drop-down menus or combo boxes) display a list of choices when you open them with a button-press. When closed, a pop-up menu indicates the current choice. Pop-up menus are useful when you want to provide a number of mutually exclusive choices, but do not want to take up the amount of space that a group of radio buttons requires.
- `'pushbutton'` – Push buttons generate an action when activated. Left-click a push button to activate it. The button appears to depress until you release the mouse button. The callback activates when you release the mouse button while still pointing within the push button.
- `'radiobutton'` – Radio buttons are similar to check boxes, but are intended to be mutually exclusive within a group of related radio buttons. When used this way, you can only select one radio button at any given time). To activate a radio button, click and release the mouse button over it. The easiest way to implement mutually exclusive behavior for a set of radio buttons is to place them within a `uibuttongroup`.
- `'slider'` – Sliders accept numeric input within a specific range when you move the “thumb” button along a bar. The location of the thumb indicates a numeric value, assigned to the `Value` property when you release the mouse button. You can set the minimum, maximum, and current values, and step sizes of a slider.

  Move the thumb by doing any one of the following:

  - Press the mouse button on the thumb, and drag it along the bar.

- Click in the bar or on arrow buttons located at both ends of the bar.
- Click the keyboard arrow keys when the slider is in focus.

- `'text'` – Static text boxes display lines of text. You typically use static text to label other controls, provide information to the user, or indicate values associated with a slider. Users cannot change static text interactively. Static text controls do not activate callback routines when clicked.
- `'togglebutton'` – Toggle buttons are similar in appearance to push buttons, but they visually indicate their state, either `on` (pressed down) or `off` (up). Clicking a toggle button changes its state, and switches its `Value` property between `Min` and `Max`.

Tips

- To simulate activation on static text by a mouse click, set the text *object's* `Enable` property to `Inactive`, and set its `ButtonDownFcn` callback to perform an action. For example, the following code displays 'Pressed' in the Command Window when you click on the`Click me!` text in the running GUI. (Without the `set` command, it would do nothing.)

```
h = uicontrol('Style','text','String','Click Me!');
set(h,'Enable','Inactive','ButtonDownFcn',...
   'disp(''Text was clicked'')')
```

- Adding a uicontrol to a figure removes the figure toolbar when the figure `Toolbar` property is `'auto'` (which is the default). To prevent this from happening, set the `Toolbar` property to `'figure'`. You can restore the toolbar by selecting **Figure Toolbar** from the **View** menu regardless of this property setting.
- The `uicontrol` function accepts property name/property value pairs, structures, and cell arrays as input arguments and optionally returns the handle of the created object. You can also use the `set` and `get` functions to set and query property values after creating the control.
- A `uicontrol` object is a child of a figure, uipanel, or uibuttongroup. Therefore, the uicontrol object does not require an axes to exist when placed in a figure window, uipanel, or uibuttongroup.

- When you pause MATLAB and a uicontrol has focus, pressing a keyboard key does not cause MATLAB to resume. Click anywhere outside a uicontrol and then press any key. For more information, see the `pause` function.

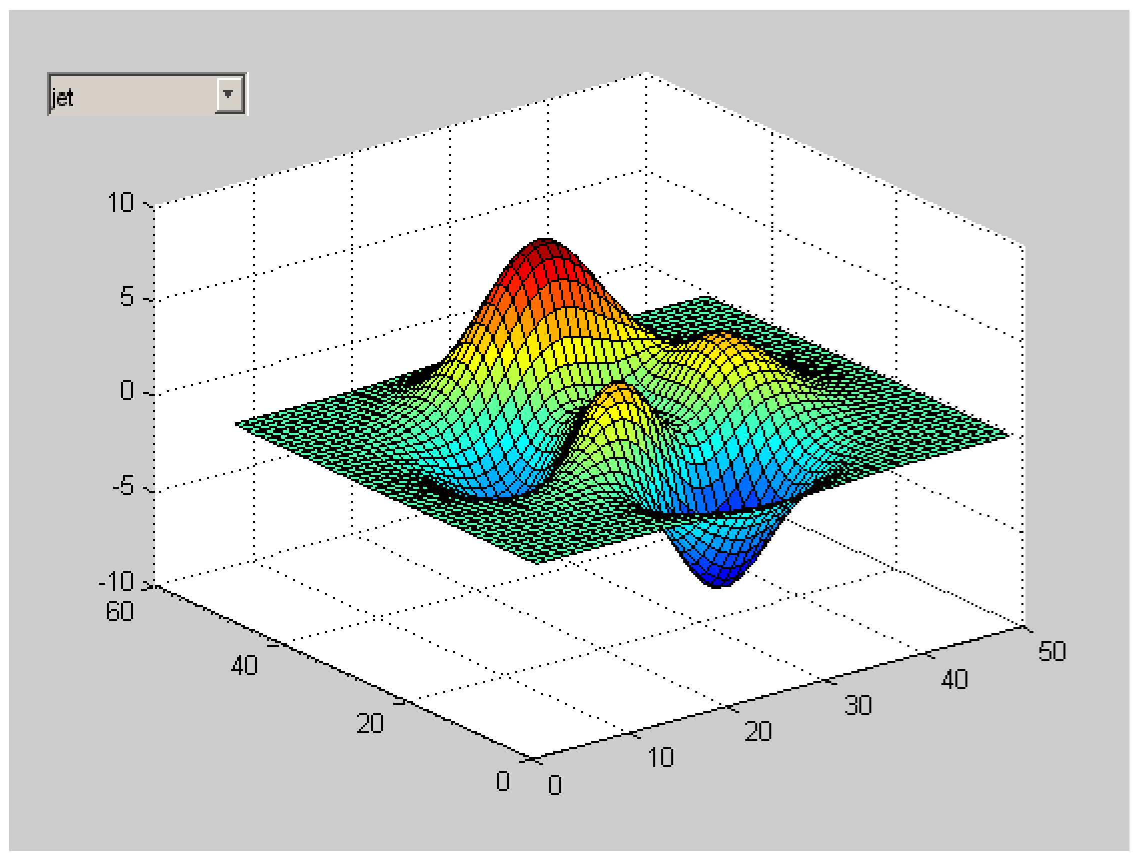
Examples

Create a figure and an axes to contain a 3-D surface plot.

```
figure
hax = axes('Units','pixels');
surf(peaks)
```

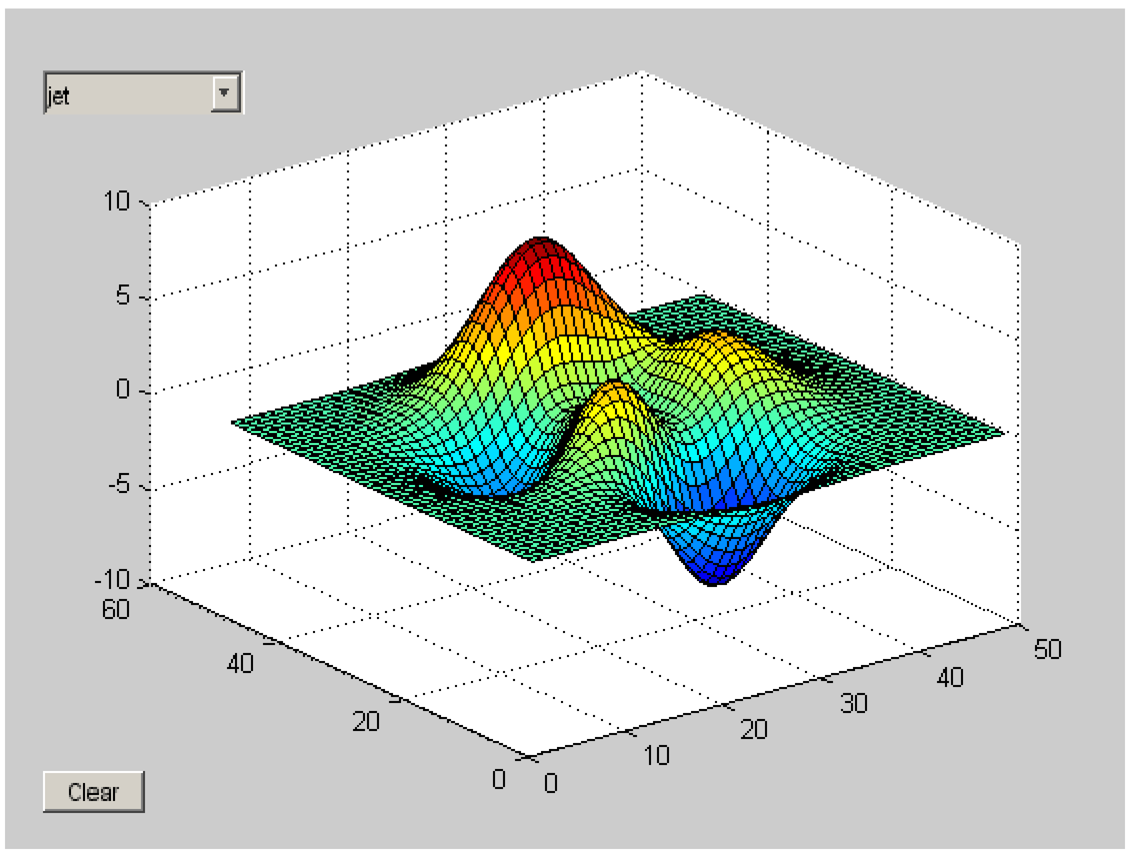
Place a uicontrol object to let users change the colormap with a pop-up menu. Supply a function handle as the object's `Callback` property:

```
uicontrol('Style', 'popup',...
          'String', 'jet|hsv|hot|cool|gray',...
          'Position', [20 340 100 50],...
          'Callback', @setmap);   % Popup function handle callback
                                  % Implemented as a local function
```



Add a different uicontrol. Create a push button that clears the current axes when pressed. Position the button inside the axes at the lower left. All uicontrols have default units of pixels. In this example, the axes has units of pixels, as well.

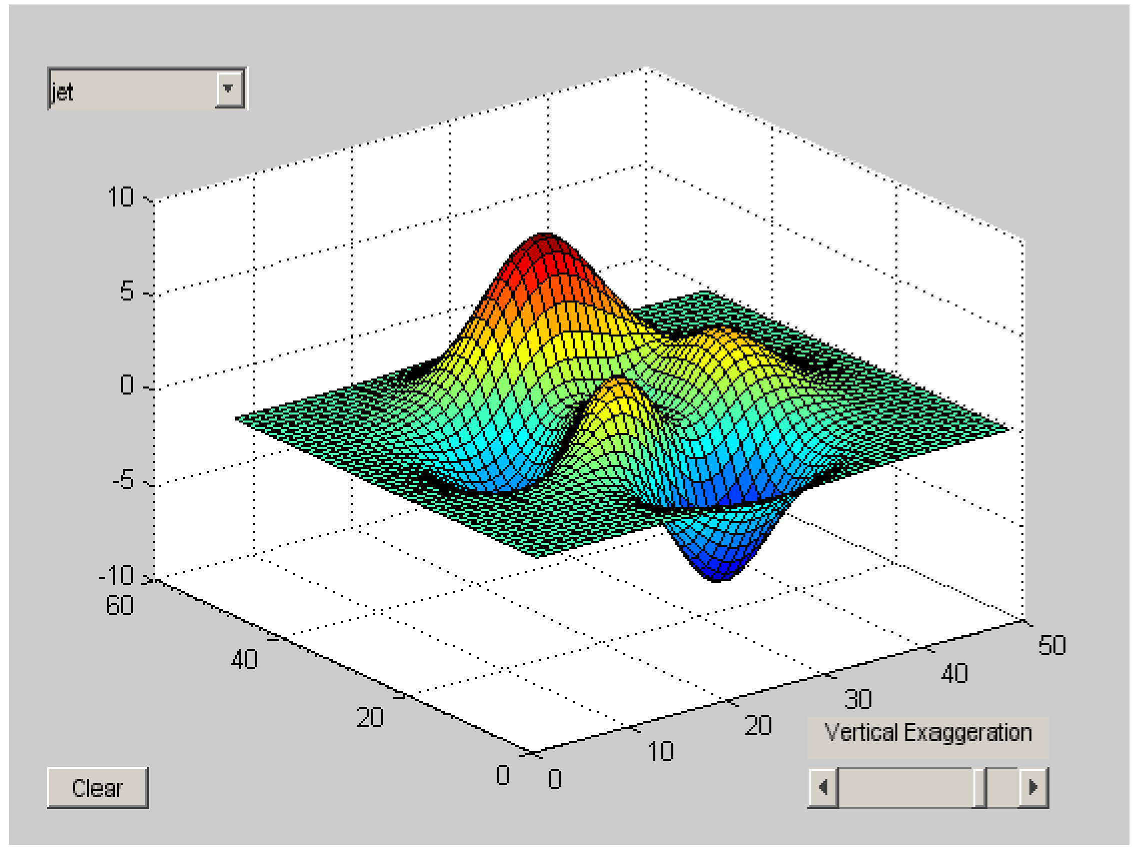
```
uicontrol('Style', 'pushbutton', 'String', 'Clear',...
        'Position', [20 20 50 20],...
        'Callback', 'cla');        % Pushbutton string callback
                                   % that calls a MATLAB function
```



Add a slider uicontrol to control the vertical scaling of the surface object. Position it at the bottom right corner of the figure. Then add a text uicontrol as a label for the slider.

```
uicontrol('Style', 'slider',...
        'Min',1,'Max',50,'Value',41,...
        'Position', [400 20 120 20],...
        'Callback', {@surfzlim,hax});
% Uses cell array function handle callback
% Implemented as a local function with an argument
```

```
uicontrol('Style','text',...
        'Position',[400 45 120 20],...
        'String','Vertical Exaggeration')
```



To operate the controls, you need callback functions. The callback for the pop-up menu is a local function called `setmap` that contains the following code.

```
function setmap(hObj,event) %#ok<INUSD>
```

```
        % Called when user activates popup menu
        val = get(hObj,'Value');
        if val ==1
            colormap(jet)
        elseif val == 2
            colormap(hsv)
        elseif val == 3
            colormap(hot)
        elseif val == 4
            colormap(cool)
        elseif val == 5
            colormap(gray)
        end
    end
```

The callback for the slider is also a local function in the same file, and contains the following code.

```
function surfzlim(hObj,event,ax) %#ok<INUSL>
    % Called to set zlim of surface in figure axes
    % when user moves the slider control
    val = 51 - get(hObj,'Value');
    zlim(ax,[-val val]);
end
```

To use the uicontrols in the example, you need to put all the functions in a MATLAB code file. Copy the following code and paste it into a new file. Save the file as `ex_uicontrol.m` on your search path, and then run it.

```
function ex_uicontrol
    % Example code for uicontrol reference page

    % Create a figure and an axes to contain a 3-D surface plot.
    figure
    hax = axes('Units','pixels');
    surf(peaks)
    % Create a uicontrol object to let users change the colormap
```

```
    % with a pop-up menu. Supply a function handle as the object's
    % Callback:
    uicontrol('Style', 'popup',...
           'String', 'hsv|hot|cool|gray',...
           'Position', [20 340 100 50],...
           'Callback', @setmap);    % Popup function handle callback
                                    % Implemented as a local function

    % Add a different uicontrol. Create a push button that clears
    % the current axes when pressed. Position the button inside
    % the axes at the lower left. All uicontrols have default units
    % of pixels. In this example, the axes does as well.
    uicontrol('Style', 'pushbutton', 'String', 'Clear',...
        'Position', [20 20 50 20],...
        'Callback', 'cla');        % Pushbutton string callback
                                   % that calls a MATLAB function

    % Add a slider uicontrol to control the vertical scaling of the
    % surface object. Position it under the Clear button.
    uicontrol('Style', 'slider',...
        'Min',1,'Max',50,'Value',41,...
        'Position', [400 20 120 20],...
        'Callback', {@surfzlim,hax});
     % Slider function handle callback
          % Implemented as a local function

    % Add a text uicontrol to label the slider.
    uicontrol('Style','text',...
        'Position',[400 45 120 20],...
        'String','Vertical Exaggeration')
end


function setmap(hObj,event) %#ok<INUSD>
    % Called when user activates popup menu
    val = get(hObj,'Value');
    if val ==1
```

```
        colormap(jet)
    elseif val == 2
        colormap(hsv)
    elseif val == 3
        colormap(hot)
    elseif val == 4
        colormap(cool)
    elseif val == 5
        colormap(gray)
    end
end

function surfzlim(hObj,event,ax) %#ok<INUSL>
    % Called to set zlim of surface in figure axes
    % when user moves the slider control
    val = 51 - get(hObj,'Value');
    zlim(ax,[-val val]);
end
```

See Also

textwrap | uibuttongroup | uimenu | uipanel | uipushtool | uitable | uitoggletool

How To

- Examples: Programming GUI Components
- "A Working GUI with Many Components"

Uicontrol Properties

Purpose

Describe user interface control (`uicontrol`) properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` commands enable you to set and query the values of properties

To change the default value of properties see “Setting Default Property Values”. You can also set default uicontrol properties on the root and figure levels:

```
set(0,'DefaultUicontrolProperty',PropertyValue...)
set(gcf,'DefaultUicontrolProperty',PropertyValue...)
```

where *Property* is the name of the uicontrol property whose default value you want to set and `PropertyValue` is the value you are specifying as the default. Use `set` and `get` to access uicontrol properties.

For information on using these uicontrols within GUIDE, the MATLAB GUI development environment, see Programming GUI Components in the MATLAB Creating GUIs documentation.

Uicontrol Properties

This section lists all properties useful to `uicontrol` objects along with valid values and descriptions of their use. Curly braces `{}` enclose default values.

Property	Purpose
`BackgroundColor`	Object background color
`BeingDeleted`	This object is being deleted
`BusyAction`	Callback routine interruption
`ButtonDownFcn`	Button-press callback routine

Property	Purpose
Callback	Control action
CData	Truecolor image displayed on the control
Children	Uicontrol objects have no children
CreateFcn	Callback routine executed during object creation
DeleteFcn	Callback routine executed during object deletion
Enable	Enable or disable the uicontrol
Extent	position rectangle (read only)
FontAngle	Character slant
FontName	Font family
FontSize	Font size
FontUnits	Font size units
FontWeight	Weight of text characters
ForegroundColor	Color of text
HandleVisibility	Whether handle is accessible from command line and GUIs
HitTest	Whether selectable by mouse click
HorizontalAlignment	Alignment of label string
Interruptible	Callback routine interruption mode
KeyPressFcn	Key press callback routine
ListboxTop	Index of top-most string displayed in list box
Max	Maximum value (depends on uicontrol object)
Min	Minimum value (depends on uicontrol object)

Property	Purpose
Parent	Uicontrol object's parent
Position	Size and location of uicontrol object
Selected	Whether object is selected
SelectionHighlight	Object highlighted when selected
SliderStep	Slider step size
String	Uicontrol object label, also list box and pop-up menu items
Style	Type of uicontrol object
Tag	User-specified object identifier
TooltipString	Content of object's tooltip
Type	Class of graphics object
UIContextMenu	Uicontextmenu object associated with the uicontrol
Units	Units to interpret position vector
UserData	User-specified data
Value	Current value of uicontrol object
Visible	Uicontrol visibility

BackgroundColor
    ColorSpec

Object background color. The color used to fill the uicontrol rectangle. Specify a color using a three-element RGB vector or one of the MATLAB predefined names. The default color is determined by system settings. See ColorSpec for more information on specifying color.

Note Platform look and feel for a GUI element can override change requests from MATLAB. It is possible your system will not honor a request to change the background color.

`BeingDeleted`
`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`
`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.

- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`
string or function handle (GUIDE sets this property)

Button-press callback routine. A callback routine that can execute when you press a mouse button while the pointer is on or near a uicontrol. Specifically:

- If the uicontrol's `Enable` property is set to `on`, the `ButtonDownFcn` callback executes when you click the right or left mouse button in a 5-pixel border around the uicontrol or when you click the right mouse button on the control itself.
- If the uicontrol's `Enable` property is set to `inactive` or `off`, the `ButtonDownFcn` executes when you click the right or left mouse button in the 5-pixel border or on the control itself.

This is useful for implementing actions to interactively modify control object properties, such as size and position, when they are clicked on (using selectmoveresize, for example).

Define this routine as a string that is a valid MATLAB expression or the name of a code file. The expression executes in the MATLAB workspace.

To add a `ButtonDownFcn` callback in GUIDE, select **View Callbacks** from the Layout Editor **View** menu, then select `ButtonDownFcn`. GUIDE sets this property to the appropriate string and adds the callback to the code file the next time you save the GUI. Alternatively, you can set this property to the string `%automatic`. The next time you save the GUI, GUIDE sets

this property to the appropriate string and adds the callback to the code file.

Use the `Callback` property to specify the callback routine that executes when you activate the enabled uicontrol (e.g., click a push button).

`Callback`
string or function handle (GUIDE sets this property)

Control action. A routine that executes whenever you activate the uicontrol object (e.g., when you click on a push button or move a slider). Define this routine as a string that is a valid MATLAB expression or the name of a code file. The expression executes in the MATLAB workspace.

For examples of `Callback` callbacks for each style of component:

- For GUIDE GUIs, see "Add Code for Components in Callbacks".
- For programmatically created GUIs, see "Examples: Program GUI Components".

`CData`
matrix

Truecolor image displayed on control. A three-dimensional matrix of RGB values that defines a truecolor image displayed on a control, which must be a **push button** or **toggle button**. Each value must be between `0.0` and `1.0`. Setting `CData` on a **radio button** or **checkbox** will replace the default `CData` on these controls. The control will continue to work as expected, but its state is not reflected by its appearance when clicked.

For **push buttons** and **toggle buttons**, `CData` overlaps the `String`. In the case of **radio buttons** and **checkboxes**, `CData` takes precedence over `String` and, depending on its size, it can displace the text.

Setting `CData` to `[]` restores the default `CData` for **radio buttons** and **checkboxes**.

`Children`
matrix

The empty matrix; `uicontrol` objects have no children.

`Clipping`
`{on} | off`

This property has no effect on `uicontrol` objects.

`CreateFcn`
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a `uicontrol` object. MATLAB sets all property values for the uicontrol before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the uicontrol being created.

Setting this property on an existing uicontrol object has no effect.

You can define a default `CreateFcn` callback for all new uicontrols. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uicontrol`. For example, the code

```
set(0,'DefaultUicontrolCreateFcn','set(gcbo,...
    ''BackgroundColor'',''white'')')
```

creates a default `CreateFcn` callback that runs whenever you create a new uicontrol. It sets the default background color of all new uicontrols.

To override this default and create a uicontrol whose `BackgroundColor` is set to a different value, call `uicontrol` with code similar to

```
hpt = uicontrol(...,'CreateFcn','set(gcbo,...
''BackgroundColor'',''blue'')')
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uicontrol` call. In the example above, if instead of redefining the `CreateFcn` property for this uicontrol, you had explicitly set `BackgroundColor` to `blue`, the default `CreateFcn` callback would have set `BackgroundColor` back to the default, i.e., `white`.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

See "Function Handle Callbacks" for information on how to use function handles to define a callback function.

`DeleteFcn`
string or function handle

Delete uicontrol callback routine. A callback routine that executes when you delete the uicontrol object (e.g., when you issue a `delete` command or clear the figure containing the uicontrol). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root CallbackObject property, which you can query using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define a callback function.

`Enable`
`{on} | inactive | off`

Enable or disable the uicontrol. This property controls how uicontrols respond to mouse button clicks, including which callback routines execute.

- `on` – The uicontrol is operational (the default).
- `inactive` – The uicontrol is not operational, but looks the same as when `Enable` is `on`.
- `off` – The uicontrol is not operational and its image (set by the `Cdata` property) is grayed out.

When you left-click on a uicontrol whose `Enable` property is `on`, MATLAB performs these actions in this order:

1. Sets the figure `SelectionType` property.
2. Executes the uicontrol `Callback` routine, if any. (Static text components do not use callbacks.)
3. Does *not* set the figure `CurrentPoint` property and does *not* execute either the uicontrol `ButtonDownFcn` or the figure `WindowButtonDownFcn` callback.

Single-clicking or double-clicking an enabled uicontrol with the left mouse button sets the figure `SelectionType` property to `normal`, unless the uicontrol `Style` is `listbox`. For list boxes, double-clicking sets the figure `SelectionType` property to `open` on the second of the two clicks, enabling the list box callback to detect a set of multiple choices.

When you left-click on a uicontrol whose `Enable` property is `off` or `inactive`, or when you right-click a uicontrol whose `Enable` property has any value, MATLAB performs these actions in this order:

1. Sets the figure `SelectionType` property.
2. Sets the figure `CurrentPoint` property.
3. Executes the figure `WindowButtonDownFcn` callback, if provided.
4. Executes the uicontrol `ButtonDownFcn` callback, if provided.

`Extent`

position rectangle (read only)

Size of uicontrol character string. A four-element vector that defines the size and position of the character string used to label the uicontrol. It has the form:

```
[0,0,width,height]
```

The first two elements are always zero. `width` and `height` are the dimensions of the rectangle. All measurements are in units specified by the `Units` property.

Since the `Extent` property is defined in the same units as the uicontrol itself, you can use this property to determine proper sizing for the uicontrol with regard to its label. Do this by

- Defining the `String` property and selecting the font using the relevant properties.
- Getting the value of the `Extent` property.
- Defining the `width` and `height` of the `Position` property to be somewhat larger than the `width` and `height` of the `Extent`.

For multiline strings, the `Extent` rectangle encompasses all the lines of text. For single line strings, the `height` element of the `Extent` property returned always indicates the height of a

single line, and its `width` element always indicates the width of the longest line, even if the string wraps when displayed on the control. Edit boxes are considered multiline if `Max` - `Min` > 1.

`FontAngle`
`{normal} | italic | oblique`

Character slant. MATLAB uses this property to select a font from those available on your particular system. Setting this property to `italic` or `oblique` selects a slanted version of the font, when it is available on your system.

`FontName`
string

Font family. The name of the font in which to display the String. To display and print properly, this must be a font that your system supports. The default font is system dependent.

Note MATLAB GUIs do not support the `Marlett` and `Symbol` font families.

To use a fixed-width font that looks good in any locale (and displays properly in Japan, where multibyte character sets are used), set `FontName` to the string `FixedWidth` (this string value is case sensitive):

```
set(uicontrol_handle, 'FontName', 'FixedWidth')
```

This parameter value eliminates the need to hard code the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan). A properly written MATLAB application that needs to use a fixed-width font should set `FontName` to `FixedWidth` and rely on the root `FixedWidthFontName` property to be set correctly in the end user's environment.

End users can adapt a MATLAB application to different locales or personal environments by setting the root `FixedWidthFontName` property to the appropriate value for that locale from `startup.m`. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

Tip To determine what fonts exist on your system (which can differ from the GUI user's system), use the `uisetfont` GUI to select a font and return its name and other characteristics in a MATLAB structure.

`FontSize`
size in `FontUnits`

Font size. A number specifying the size of the font in which to display the String, in units determined by the `FontUnits` property. The default point size is system dependent.

`FontUnits`
`{points} | normalized | inches |`
`centimeters | pixels`

Font size units. This property determines the units used by the `FontSize` property. `Normalized` units interpret `FontSize` as a fraction of the height of the uicontrol. When you resize the uicontrol, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = $^1/_{72}$ inch).

`FontWeight`
`light | {normal} | demi | bold`

Weight of text characters. MATLAB uses this property to select a font from those available on your particular system. Setting this property to `bold` causes MATLAB to use a bold version of the font, when it is available on your system.

`ForegroundColor`
`ColorSpec`

Color of text. This property determines the color of the text defined for the `String` property (the uicontrol label). Specify a color using a three-element RGB vector or one of the MATLAB predefined names. The default text color is black. See `ColorSpec` for more information on specifying color.

Note If you change the `ForegroundColor` for a uicontrol `listbox`, MATLAB uses that color for all `listbox` items *except* selected `listbox` items. For selected items MATLAB uses a color that ensures good contrast between the text of selected items and the color that indicates a selection.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from

command-line users, while allowing callback routines to have complete access to object handles.

- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to on to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

Note Radio buttons and toggle buttons that are managed by a uibuttongroup should not be accessed outside the button group. Set the `HandleVisibility` of such radio buttons and toggle buttons to `off` to prevent inadvertent access.

`HitTest`
`{on} | off`

Selectable by mouse click. This property has no effect on uicontrol objects.

`HorizontalAlignment`
`left | {center} | right`

Horizontal alignment of label string. This property determines the justification of the text defined for the `String` property (the uicontrol label):

- `left` — Text is left justified with respect to the uicontrol.
- `center` — Text is centered with respect to the uicontrol.
- `right` — Text is right justified with respect to the uicontrol.

The contents of `edit` and `text` uicontrols are always vertically aligned to the top of the rectangle.

Note On Microsoft Windows systems, although this property can be specified on all uicontrols, it affects only `edit` and `text` uicontrols.

`Interruptible`

`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For user interface objects, the `Interruptible` property affects the callbacks for these properties only:

- `ButtonDownFcn`
- `KeyPressFcn`
- `KeyReleaseFcn`
- `WindowButtonDownFcn`
- `WindowButtonMotionFcn`
- `WindowButtonUpFcn`
- `WindowKeyPressFcn`
- `WindowKeyReleaseFcn`
- `WindowScrollWheelFcn`

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

The `BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting the `Interruptible` property to `on` (default), allows a callback from other user interface objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see “Control Callback Execution and Interruption”.

Note If the interrupting callback is a `DeleteFcn`or `CreateFcn` callback, or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. An object's `ButtonDownFcn` or `Callback` routine is processed according to the rules described previously in this section.

`KeyPressFcn`
string or function handle

Key press callback function. A callback routine invoked by a key press when the callback's uicontrol object has focus. Focus is denoted by a border or a dotted border, respectively, in UNIX and Microsoft Windows. If no uicontrol has focus, the figure's key press callback function, if any, is invoked. `KeyPressFcn` can be a function handle, the name of a code file, or any legal MATLAB expression.

If the specified value is the name of a code file, the callback routine can query the figure's `CurrentCharacter` property to determine what particular key was pressed and thereby limit the callback execution to specific keys.

If the specified value is a function handle, the callback routine can retrieve information about the key that was pressed from its event data structure argument.

Event Data Structure Field	Description	Examples:			
		a	=	Shift	Shift/a
Character	Character interpretation of the key that was pressed.	'a'	'='	''	'A'
Modifier	Current modifier, such as 'control', or an empty cell array if there is no modifier	{1x0 cell}	{1x0 cell}	{'shift'}	{'shift'}
Key	Name of the key that was pressed.	'a'	'equal'	'shift'	'a'

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

ListboxTop
scalar

Index of top-most string displayed in list box. This property applies only to the listbox style of uicontrol. It specifies which string appears in the top-most position in a list box that is not large enough to display all list entries. ListboxTop is an index into the array of strings defined by the String property and must have a value between 1 and the number of strings. Noninteger values are fixed to the next lowest integer.

Max
scalar

Maximum value. This property specifies the largest value allowed for the Value property. Different styles of uicontrols interpret Max differently:

- Check boxes – Max is the setting of the Value property while the check box is selected.
- Editable text – The Value property does not apply. If Max - Min > 1, then editable text boxes accept multiline input. If Max - Min

<= 1, then editable text boxes accept only single line input. The absolute values of `Max` and `Min` have no effect on the number of lines an edit box can contain; a multiline edit box can contain any number of lines.

- List boxes – If `Max - Min` > 1, then list boxes allow multiple item selection. If `Max - Min` <= 1, then list boxes do not allow multiple item selection. When they do, `Value` can be a vector of indices.
- Radio buttons – `Max` is the setting of the `Value` property when the radio button is selected.
- Sliders – `Max` is the maximum slider value and must be greater than the `Min` property. The default is 1.
- Toggle buttons – `Max` is the value of the `Value` property when the toggle button is selected. The default is 1.
- Pop-up menus, push buttons, and static text do not use the `Max` property.

`Min`

scalar

Minimum value. This property specifies the smallest value allowed for the `Value` property. Different styles of uicontrols interpret `Min` differently:

- Check boxes – `Min` is the setting of the `Value` property while the check box is not selected.
- Editable text – The `Value` property does not apply. If `Max - Min` > 1, then editable text boxes accept multiline input. If `Max - Min` <= 1, then editable text boxes accept only single line input. The absolute values of `Max` and `Min` have no effect on the number of lines an edit box can contain; a multiline edit box can contain any number of lines.
- List boxes – If `Max - Min` > 1, then list boxes allow multiple item selection. If `Max - Min` <= 1, then list boxes allow only single item selection. When they do, `Value` can be a vector of indices.

- Radio buttons – `Min` is the setting of the `Value` property when the radio button is not selected.
- Sliders – `Min` is the minimum slider value and must be less than `Max`. The default is 0.
- Toggle buttons – `Min` is the value of the `Value` property when the toggle button is not selected. The default is 0.
- Pop-up menus, push buttons, and static text do not use the `Min` property.

`Parent`

handle

Uicontrol parent. The handle of the uicontrol's parent object. You can move a uicontrol object to another figure, uipanel, or uibuttongroup by setting this property to the handle of the new parent.

`Position`

position rectangle

Size and location of uicontrol. The rectangle defined by this property specifies the size and location of the control within the parent figure window, uipanel, or uibuttongroup. Specify `Position` as

```
[left bottom width height]
```

`left` and `bottom` are the distance from the lower-left corner of the parent object to the lower-left corner of the uicontrol object. `width` and `height` are the dimensions of the uicontrol rectangle. All measurements are in units specified by the `Units` property.

On Microsoft Windows systems, the height of pop-up menus is automatically determined by the size of the font. The value you specify for the `height` of the `Position` property has no effect.

The `width` and `height` values determine the orientation of sliders. If `width` is greater than `height`, then the slider is oriented horizontally, If `height` is greater than `width`, then the slider is oriented vertically.

Note The height of a pop-up menu is determined by the font size. The height you set in the position vector is ignored. The height element of the position vector is not changed.

On Mac platforms, the height of a horizontal slider is constrained. If the height you set in the position vector exceeds this constraint, the displayed height of the slider is the maximum allowed. The height element of the position vector is not changed.

`Selected`

`on | {off}` (read only)

Is object selected. When this property is `on`, MATLAB displays selection handles if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`

`{on} | off`

Object highlight when selected. When the `Selected` property is on, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`SliderStep`

`[minor_step major_step]`

Slider step size, a two-element vector of positive values that indicates the size of the major and minor steps as a percent change in slider value. Both steps should be greater than `1e-6`,

and `minor_step` should be less than or equal to `major_step`. The slider `Value` changes up or down by `minor_step` when you click the arrow button, and up or down by `major_step` when you click the slider trough, also called the channel.

The actual step size is a function of the specified `SliderStep` and the total slider range (`Max - Min`). The default, `[0.01 0.10]`, provides a 1 percent change for clicks on the arrow button and a 10 percent change for clicks in the trough.

For example, if you create the following slider,

```
uicontrol('Style','slider','Min',1,'Max',7,...
         'Value',2,'SliderStep',[0.1 0.6])
```

clicking an arrow button moves the thumb indicator toward it by `minor_step*(max-min)`:

```
0.1*(7-1)
ans =
    0.6000
```

Clicking the trough moves the thumb indicator toward the mouse pointer by `major_step*(max-min)`:

```
0.6*(7-1)
ans =
    3.6000
```

If clicking moves the slider to a value outside the range, the thumb indicator moves only to the `Max` or `Min` value.

As `major_step` increases, the slider thumb indicator grows longer. When `major_step` is 1, the thumb is half as long as the trough. When `major_step` is greater than 1, the thumb continues to grow, slowly approaching the full length of the trough.

See also the `Max`, `Min`, and `Value` properties.

`String`

string

Uicontrol label, list box items, pop-up menu choices.

For check boxes, editable text, push buttons, radio buttons, static text, and toggle buttons, the text displayed on the object. For list boxes and pop-up menus, the set of entries or items displayed in the object.

Note If you specify a numerical value for `String`, MATLAB converts it to `char` but the result may not be what you expect. If you have numerical data, you should first convert it to a string, e.g., using `num2str`, before assigning it to the `String` property.

For uicontrol objects that display only one line of text (check box, push button, radio button, toggle button), if the string value is specified as a cell array of strings or padded string matrix, only the first string of a cell array or of a padded string matrix is displayed; the rest are ignored. Vertical slash ('`|`') characters are not interpreted as line breaks and instead show up in the text displayed in the uicontrol.

For multiple line editable text or static text controls, line breaks occur between each row of the string matrix, and each cell of a cell array of strings. Vertical slash ('`|`') characters and `\n` characters are not interpreted as line breaks, and instead show up in the text displayed in the uicontrol. Text in edit text and static text boxes is always vertically top-aligned. For **editable text**, the `String` property value is set to the string entered by the user when the control loses focus. The `String` property for **static text** can only be set programmatically (or when authoring a GUI in GUIDE).

To place multiple items on a list box or pop-up menu, you can specify the items in any of the formats shown in the following table.

String Property Format	Example
Cell array of strings	`{'one' 'two' 'three'}`
Padded string column matrix	`['one  ';'two  ';'three']`
String vector using vertical slash (|) as a line separator	`['one|two|three']`

If you specify a component width that is too small to accommodate one or more of the specified strings, MATLAB truncates those strings with an ellipsis. Use the `Value` property to set the index of the initial item selected.

For **check boxes**, **push buttons**, **radio buttons**, **toggle buttons**, and the selected item in **popup menus**, when the specified text is clipped because it is too long for the uicontrol, an ellipsis (...) is appended to the text in the active GUI to indicate that it has been clipped.

For **push buttons** and **toggle buttons**, `CData` overlaps the `String`. In the case of **radio buttons** and **checkboxes**, `CData` takes precedence over `String` and, depending on its size, can displace the text.

Reserved Words There are three reserved words: `default`, `remove`, `factory` (case sensitive). If you want to use one of these reserved words in the `String` property, you must precede it with a backslash (`'\'`) character. For example,

```
h = uicontrol('Style','edit','String','\default');
```

`Style`

`{pushbutton} | togglebutton | radiobutton | checkbox | edit | text | slider | frame | listbox | popupmenu`

Style of uicontrol object to create. The `Style` property specifies the kind of uicontrol to create. See the `uicontrol` Description section for information on each type.

`Tag`

string (GUIDE sets this property)

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

`TooltipString`

string

Content of tooltip for object. The `TooltipString` property specifies the text of the tooltip associated with the uicontrol. When the user moves the mouse pointer over the control and leaves it there, the tooltip is displayed.

To create a tooltip that has more than one line of text, use `sprintf` to generate a string containing newline (`\n`) characters and then set the `TooltipString` to that value. For example:

```
h = uicontrol('Style','pushbutton');
s = sprintf('Button tooltip line 1\nButton tooltip line 2');
set(h,'TooltipString',s)
```

`Type`

string (read only)

Class of graphics object. For uicontrol objects, `Type` is always the string `'uicontrol'`.

`UIContextMenu`

handle

Associate a context menu with uicontrol. Assign this property the handle of a `uicontextmenu` object. MATLAB displays the context menu whenever you right-click over the uicontrol. Use the `uicontextmenu` function to create the context menu.

`Units`

`{pixels} | normalized | inches | centimeters | points | characters` (GUIDE default: `normalized`)

Units of measurement. MATLAB uses these units to interpret the `Extent` and `Position` properties. All units are measured from the lower-left corner of the parent object.

- `Normalized` units map the lower-left corner of the parent object to (0,0) and the upper-right corner to (1.0,1.0).
- `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).
- `Character` units are characters using the default system font; the width of one character is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is set to the default value.

`UserData`
matrix

User-specified data. Any data you want to associate with the uicontrol object. MATLAB does not use this data, but you can access it using `set` and `get`.

`Value`
scalar or vector

Current value of uicontrol. The uicontrol style determines the possible values this property can have:

- Check boxes set `Value` to `Max` when they are on (when selected) and `Min` when off (not selected).
- List boxes set `Value` to a vector of indices corresponding to the selected list entries, where 1 corresponds to the first item in the list.
- Pop-up menus set `Value` to the index of the item selected, where 1 corresponds to the first item in the menu. The Examples section shows how to use the `Value` property to determine which item has been selected.
- Radio buttons set `Value` to `Max` when they are on (when selected) and `Min` when off (not selected).
- Sliders set `Value` to the number indicated by the slider bar.
- Toggle buttons set `Value` to `Max` when they are down (selected) and `Min` when up (not selected).
- Editable text, push buttons, and static text do not set this property.

Set the `Value` property either interactively with the mouse or through a call to the `set` function. The display reflects changes made to `Value`.

`Visible`
`{on} | off`

Uicontrol visibility. By default, all uicontrols are visible. When set to `off`, the uicontrol is not visible, but still exists and you can query and set its properties.

Note Setting `Visible` to `off` for uicontrols that are not displayed initially in the GUI, can result in faster startup time for the GUI.

---

uigetdir

Purpose

Open standard dialog box for selecting directory

Syntax

```
folder_name = uigetdir
folder_name = uigetdir(start_path)
folder_name = uigetdir(start_path,dialog_title)
```

Description

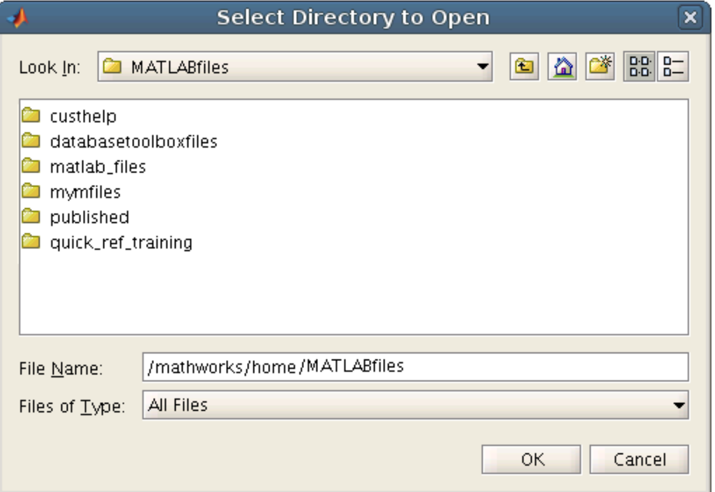
`folder_name = uigetdir` displays a modal dialog box enabling you to navigate the folder hierarchy and select a folder or type the name of a folder. If the folder exists, `uigetdir` returns the selected path when you click **OK**. If you type the name of a folder that does not exist, `uigetdir` returns the name of the current folder. If you click **Cancel**, or close the dialog box, `uigetdir` returns `0`. On Microsoft Windows platforms, `uigetdir` opens a dialog box in the base folder (the Windows desktop) with the current folder selected.

`folder_name = uigetdir(start_path)`opens a dialog box with the folder specified by `start_path` selected. If `start_path` is a valid path, the dialog box opens in the specified folder. If `start_path` is an empty string (`''`) or is not a valid path, the dialog box opens in the current folder.

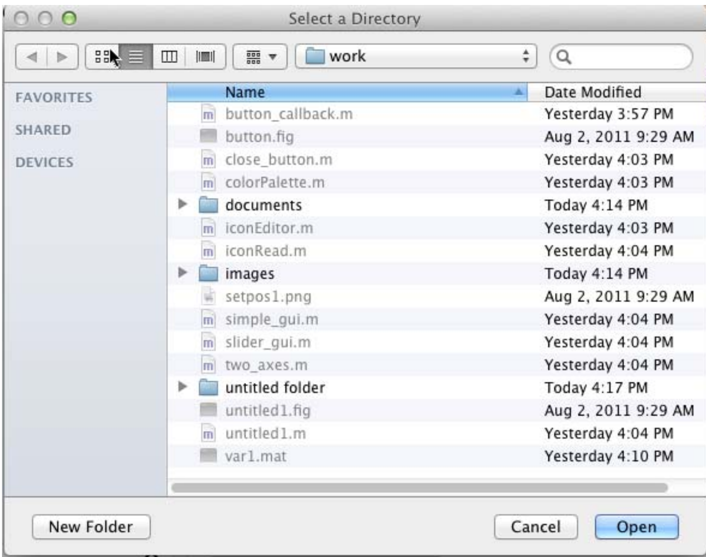
`folder_name = uigetdir(start_path,dialog_title)` opens a dialog box with the specified title. On Windows and UNIX platforms, the string replaces the default caption inside the dialog box for specifying instructions to the user. The default `dialog_title` is **Select folder to Open**.

On Windows platforms, you can click the **New Folder** button to add a new folder to the folder hierarchy displayed. You can also drag and drop existing directories into different folders.

On UNIX platforms, `uigetdir` opens a dialog box in the startup folder (the one you are in when you start MATLAB), with the current directory selected. The `dialog_title` string replaces the default title of the dialog box. The dialog box looks similar to the following figure.



On Mac platforms, `uigetdir` opens a dialog box in the startup folder (the one you are in when you start MATLAB), with the current directory selected. The dialog box looks similar to the following figure.



Note A modal dialog box prevents you from interacting with other MATLAB windows before responding. To block MATLAB program execution as well, use the `uiwait` function. For more information about modal dialog boxes, see `WindowStyle` in the MATLAB Figure Properties.

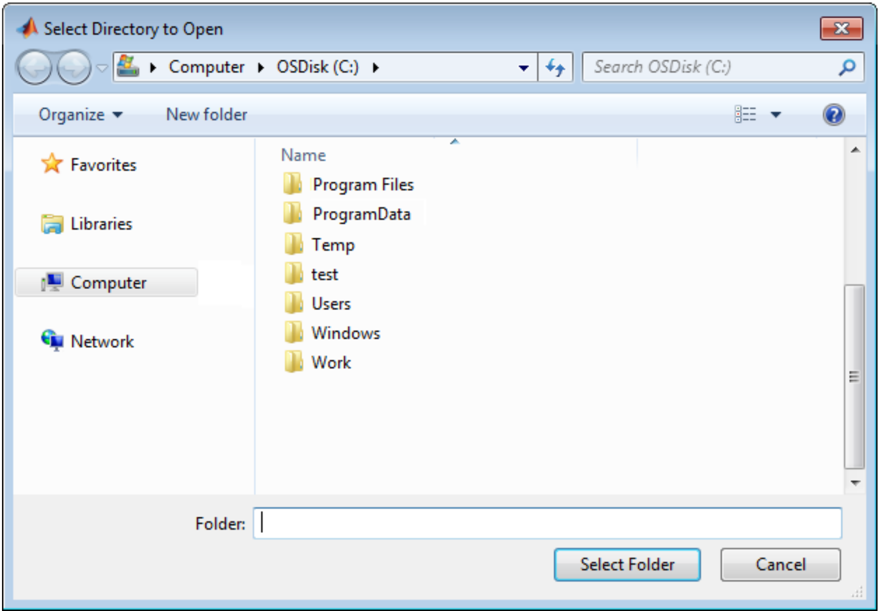
The `pwd` and `cd` functions return the name of the current folder.

Examples

The following statement displays directories on the `C:` drive.

```
dname = uigetdir('C:\');
```

A dialog box such as the following displays (on Windows).



If you double-click the `Program Files` folder, and the `MATLAB` subfolder, and then click **Select Folder**, `uigetdir` returns `C:\Program Files\MATLAB`to `dname`:

The following statement uses `matlabroot` to display the MATLAB root folder in the dialog box:

```
uigetdir(matlabroot,'MATLAB Root Directory')
```

Assuming that MATLAB is installed on drive `C:\`, selecting the folder `MATLAB\R2012b\notebook\pc` from the dialog box, returns this string:

```
C:\Program Files\MATLAB\R2012b\notebook\pc
```

See Also

`uigetfile` | `uiputfile`

uigetfile

Purpose

Open standard dialog box for retrieving files

Syntax

```
filename = uigetfile
[FileName,PathName,FilterIndex] = uigetfile(FilterSpec)
[FileName,PathName,FilterIndex] = uigetfile(FilterSpec,
   DialogTitle)
[FileName,PathName,FilterIndex] = uigetfile(FilterSpec,
   DialogTitle,DefaultName)
[FileName,PathName,FilterIndex] = uigetfile(...,'MultiSelect',
   selectmode)
```

Description

`filename = uigetfile` displays a modal dialog box that lists files in the current folder and enables you to select or enter the name of a file. If the file name is valid and the file exists, `uigetfile` returns the file name as a string when you click **Open**. Otherwise `uigetfile` displays an appropriate error message, after which control returns to the dialog box. You can then enter another file name or click **Cancel**. If you click **Cancel** or close the dialog window, `uigetfile` returns `0`.

Note Successful execution of `uigetfile` does not open a file; it only returns the name of an existing file that you identify.

`[FileName,PathName,FilterIndex] = uigetfile(FilterSpec)` displays only those files with extensions that match `FilterSpec`. On some platforms `uigetfile` also displays in gray the files that do not match `FilterSpec`. The `uigetfile` function appends `'All Files'` to the list of file types. `FilterSpec` can be a string or a cell array of strings, and can include the `*` wildcard.

- If `FilterSpec` is a file name, that file name displays, selected in the **File name** field. The extension of the file is the default filter.
- `FilterSpec` can include a path. That path can contain `'.'`,`'..'`, `\`, `'/'`, or `'~'`. For example, `'../*.m'` lists all code files in the folder above the current folder.

- If FilterSpec is a folder name, uigetfile displays the contents of that folder, the **File name** field is empty, and no filter applies. To specify a folder name, make the last character of FilterSpec either '\' or '/'.
- If FilterSpec is a cell array of strings, it can include two columns. The first column contains a list of file extensions. The optional second column contains a corresponding list of descriptions. These descriptions replace standard descriptions in the **Files of type** field. A description cannot be an empty string. The second and third examples illustrate use of a cell array as FilterSpec.

If FilterSpec is missing or empty, uigetfile uses the default list of file types (for example, all MATLAB files).

After you click **Open** and if the file name exists,uigetfile returns the name of the file in FileName and its path in PathName. If you click **Cancel** or close the dialog window, the function sets FileName and PathName to 0.

FilterIndex is the index of the filter selected in the dialog box. Indexing starts at 1. If you click **Cancel** or close the dialog window, the function sets FilterIndex to 0.

[FileName,PathName,FilterIndex] = uigetfile(FilterSpec,DialogTitle)displays a dialog box that has the title DialogTitle. To use the default file types and specify a dialog title, enter

```
uigetfile('',DialogTitle)
```

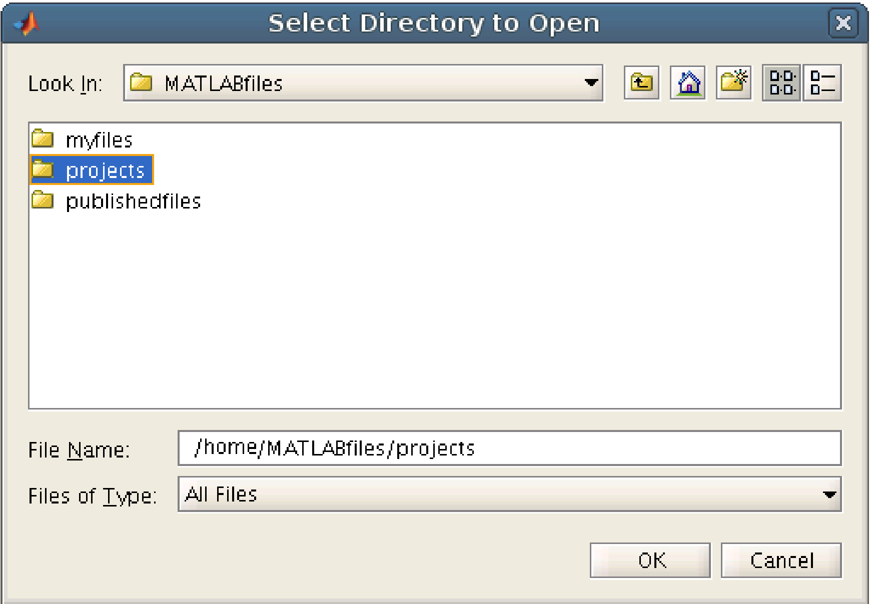
[FileName,PathName,FilterIndex] = uigetfile(FilterSpec,DialogTitle,DefaultName)displays a dialog box in which the file name specified by DefaultName appears in the **File name** field. DefaultName can also be a path or a path/filename. In this case, uigetfile opens the dialog box in the folder specified by the path. You can use '.','..', \, or '/' in the DefaultName argument. To specify a folder name, make the last character of DefaultName either '\' or '/'. If the specified path does not exist, uigetfile opens the dialog box in the current folder.

`[FileName,PathName,FilterIndex] = uigetfile(...,'MultiSelect',selectmode)` opens the dialog box in *multiselect* mode. Valid values for `selectmode` are `'on'` and `'off'` (the default, which allows single selection only). If `'MultiSelect'` is `'on'` and you select more than one file in the dialog box, then `FileName` is a cell array of strings. Each array element contains the name of a selected file. File names in the cell array are sorted in the order your platform uses. If you select multiple files, they must be in the same folder, otherwise MATLAB displays a warning dialog box. Be aware that Microsoft Windows libraries can span multiple folders. `PathName` is a string identifying the folder containing the files.

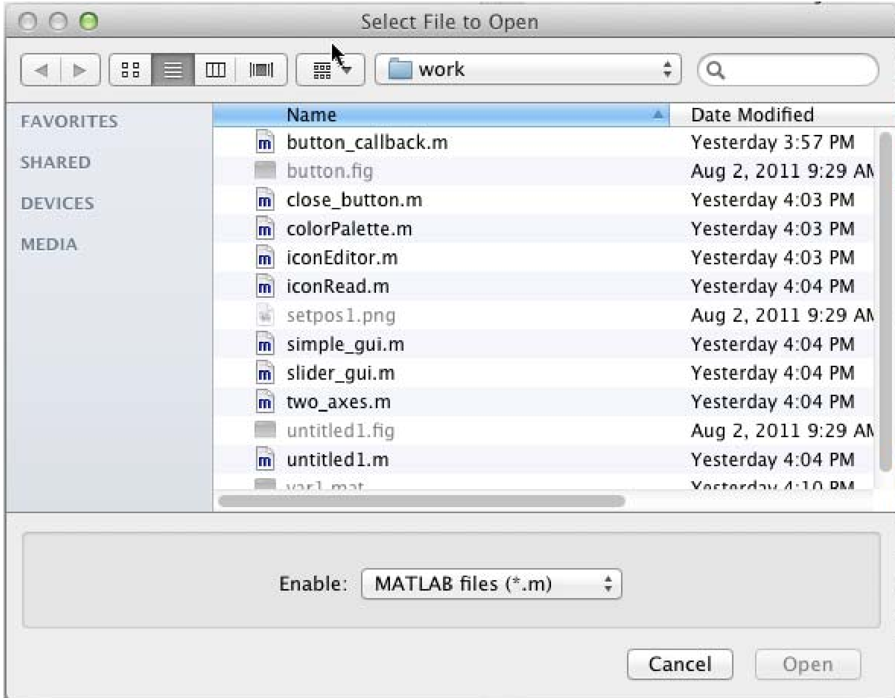
If you include either of the wildcard characters, `'*'` or `'?'`, in a file name, `uigetfile` does not respond to clicking **Open**. The dialog box remains open until you cancel it or remove the wildcard characters. This restriction applies to all platforms, even to file systems that permit these characters in file names.

For Windows platforms, the dialog box is the Windows dialog box native to your platform. Depending on your version of Windows, dialogs you see can differ from the figures shown in following examples.

For UNIX platforms, the dialog box is like the one shown in the following figure.



For Mac platforms, the dialog box is like the one shown in the following figure.



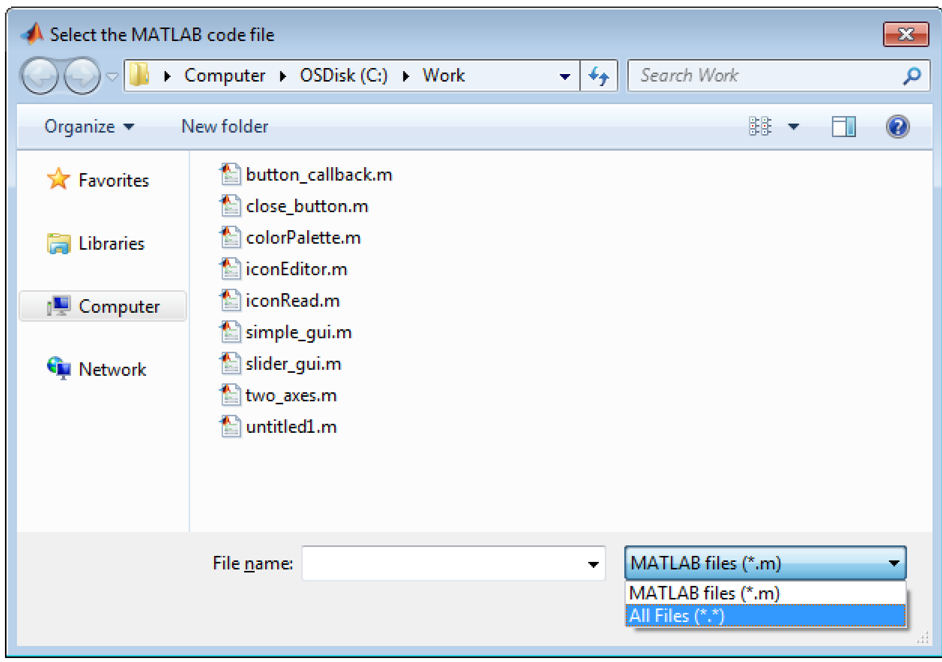
Note A modal dialog box prevents you from interacting with other windows before responding. To block MATLAB program execution, use the `uiwait` function. For more information about modal dialog boxes, see `WindowStyle` in the MATLAB Figure Properties.

Examples

The following statement displays a dialog box for retrieving a file. The dialog box lists all MATLAB code files within a selected directory. `uigetfile` returns the name and path of the selected file in `FileName` and `PathName`. `uigetfile` appends `All Files(*.*)` to the file types when `FilterSpec` is a string.

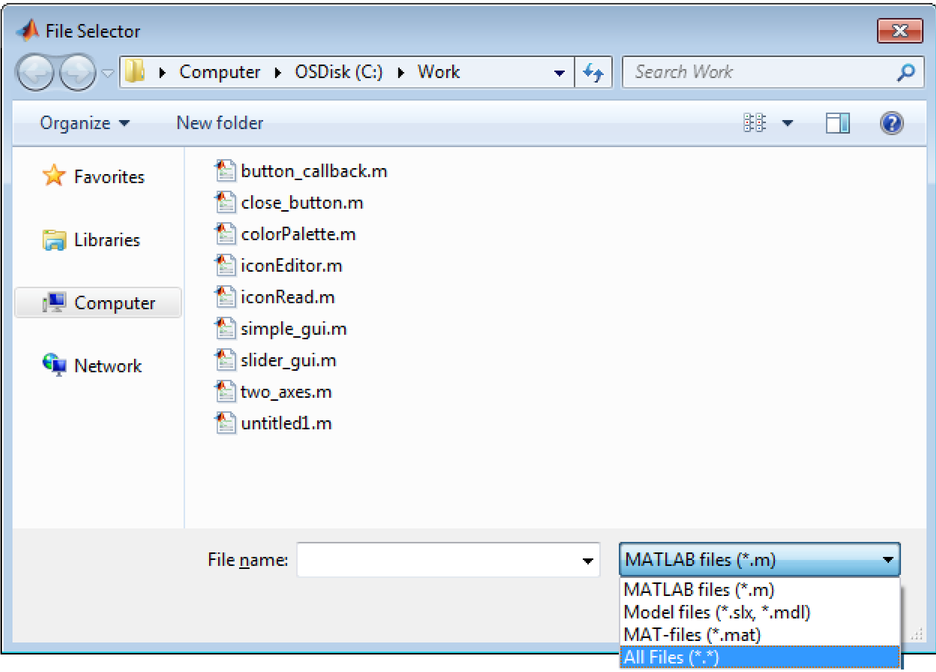
```
[FileName,PathName] = uigetfile('*.m','Select the MATLAB code file');
```

The following figure shows the dialog box with the file type drop-down list open.



To create a list of file types that appears in the file type drop-down list, separate the file extensions with semicolons, as in the following code. `uigetfile` displays a default description for each known file type, such as "Model files" for Simulink `.slx` and `.mdl` files.

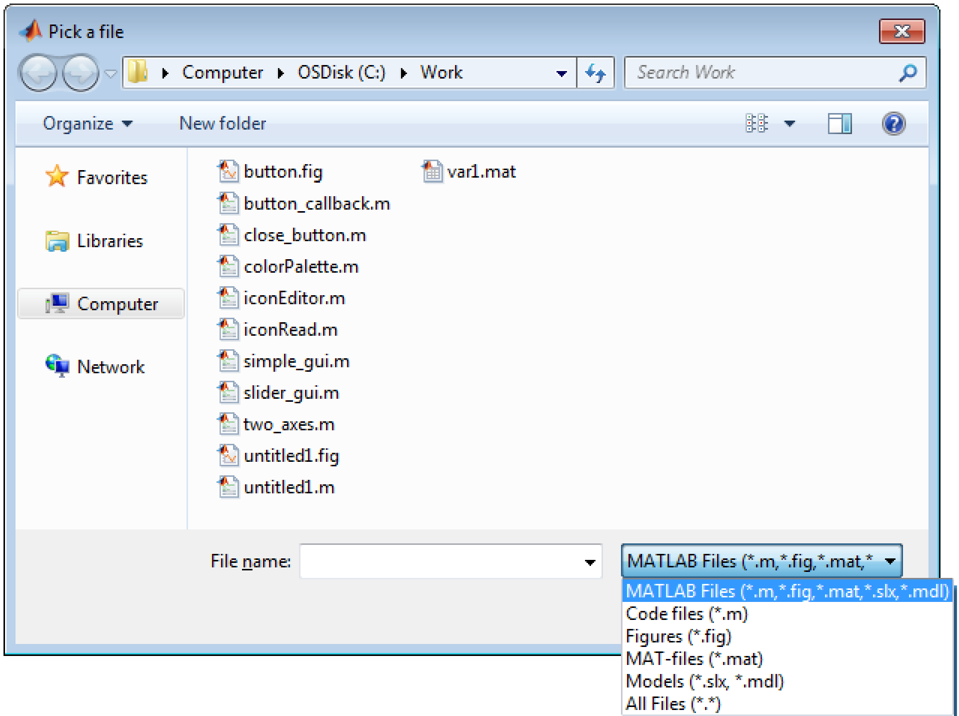
```
[filename, pathname] = ...
     uigetfile({'*.m';'*.slx';'*.mat';'*.*'},'File Selector');
```



If you want to create a list of file types and give them descriptions that are different from the defaults, use a cell array, as in the following code. This example also associates multiple file types with the `'MATLAB Files'` and `'Models'` descriptions.

```
[filename, pathname] = uigetfile( ...
{'*.m;*.fig;*.mat;*.slx;*.mdl','MATLAB Files (*.m,*.fig,*.mat,*.slx,*.mdl)';
   '*.m',  'Code files (*.m)'; ...
   '*.fig','Figures (*.fig)'; ...
   '*.mat','MAT-files (*.mat)'; ...
   '*.mdl;*.slx','Models (*.slx, *.mdl)'; ...
   '*.*',  'All Files (*.*)'}, ...
   'Pick a file');
```

The first column of the cell array contains the file extensions, while the second contains your descriptions of the file types. In this example, the first entry of column one contains several extensions, separated by semicolons, which are all associated with the description `'MATLAB Files (*.m,*.fig,*.mat,*.mdl)'`. The code produces the dialog box shown in the following figure.



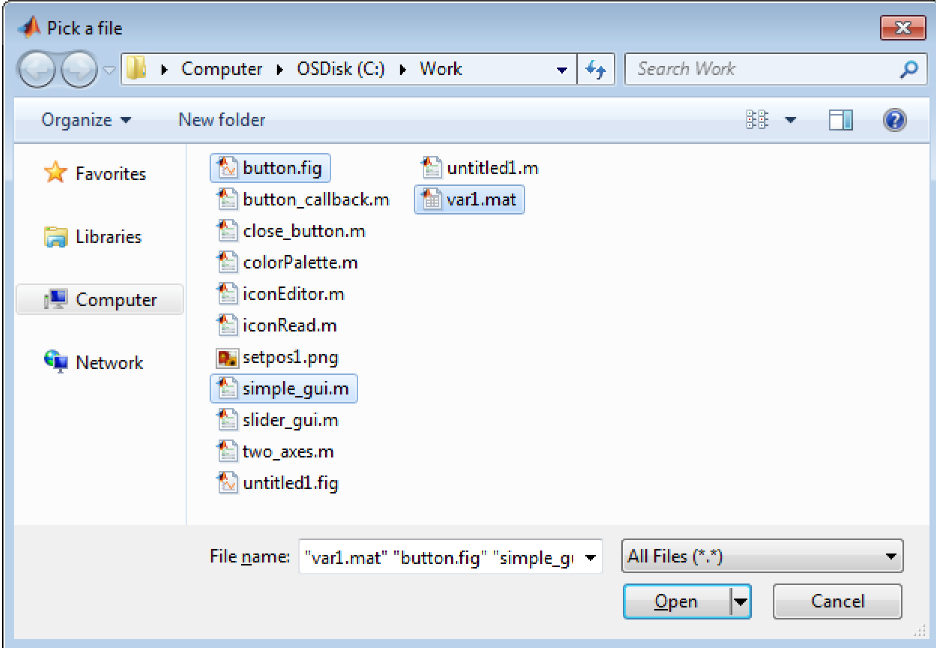
The following code lets you select a file and then displays a message in the Command Window that summarizes the result.

```
[filename, pathname] = uigetfile('*.m', 'Select a MATLAB code file');
if isequal(filename,0)
   disp('User selected Cancel')
```

```
else
   disp(['User selected ', fullfile(pathname, filename)])
end
```

This code creates a list of file types and gives them descriptions that are different from the defaults. It also enables multiple-file selection. Select multiple files by holding down the **Shift** or **Ctrl** key and clicking on additional file names.

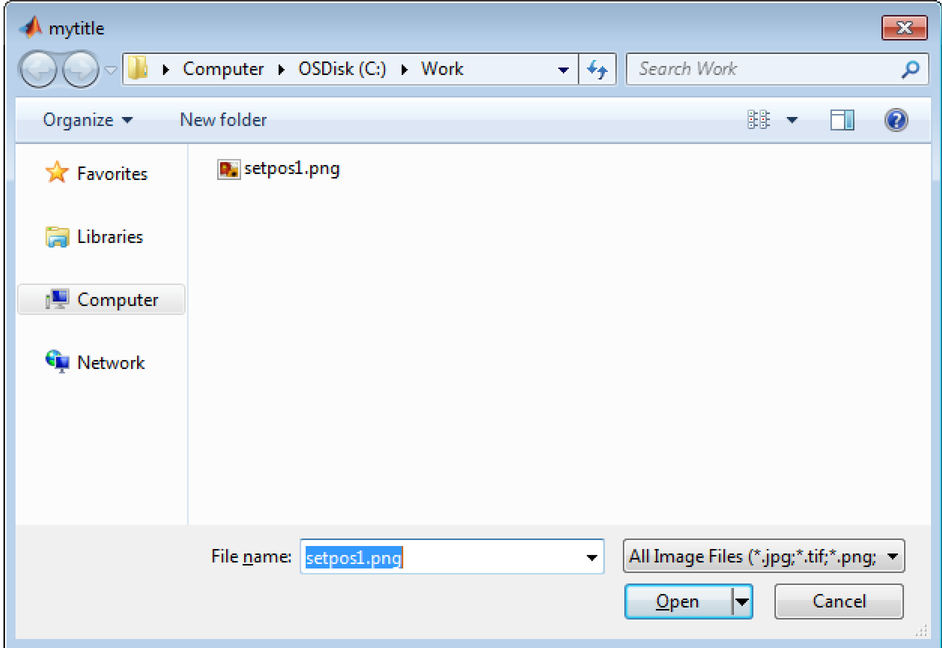
```
[filename, pathname, filterindex] = uigetfile( ...
{  '*.mat','MAT-files (*.mat)'; ...
   '*.slx;*.mdl','Models (*.slx, *.mdl)'; ...
   '*.*',  'All Files (*.*)'}, ...
   'Pick a file', ...
   'MultiSelect', 'on');
```



As mentioned previously, `uigetfile` does not open the file or files you select.

You can use the `DefaultName` argument to specify a start path and a default file name for the dialog box.

```
uigetfile({'*.jpg;*.tif;*.png;*.gif','All Image Files';...
          '*.*','All Files' },'mytitle',...
          'C:\myfiles\my_examples\gbtools\setpos1.png')
```



Alternatives

Use the `dir` function to return a filtered or unfiltered list of files in your current folder or a folder you specify. `dir` also can return file attributes.

See Also

`uigetdir` | `uiopen` | `uiputfile`

Purpose

Specify and conditionally open dialog box according to user preference

Syntax

```
pref_value = uigetpref(group,pref,title,question,
   pref_choices)
[pref_value,dlgshown] = uigetpref(...)
[...] = uigetpref(... 'Name',value)
```

Description

`pref_value = uigetpref(group,pref,title,question,pref_choices)` returns one of the strings in `pref_choices` in either of two ways:

- Displays a multiple-choice dialog box that prompts you to answer a question. The dialog box includes a check box with the label **Do not show this dialog again**.
- Retrieves a previous answer from the preferences data base and returns it without displaying a dialog. No dialog is displayed if you previously selected the check box **Do not show this dialog again**.

`[pref_value,dlgshown] = uigetpref(...)` also returns in `dlgshown` whether or not the dialog displayed.

`[...]  = uigetpref(...  'Name',value)` specify optional name-value pairs to control how dialogs display.

Tips

- You must supply all input arguments up to and including `pref_choices`.
- `uigetpref` creates specified groups and preferences, if they do not currently exist. To delete a preference group you no longer need, use `rmpref`.
- The string returned in `pref_value` is a preference name (as specified in `pref`), not its button label (as specified in `pref_choices`).
- After you select the check box **Do not show this dialog again** and close the dialog box, the dialog box does not display again for the same preference. To reenable dialogs that are being suppressed by preferences, use the command:

```
uisetpref('clearall')
```

Executing `uisetpref` with this command re-enables *all* preference dialogs you have defined with `uigetpref`, not just the most recent one.

Input Arguments

group

String specifying the name of the preference group that preference `pref` belongs to. If the group does not already exist, `uigetpref` creates it.

Default: None

pref

String specifying the name of the preference that controls whether the dialog displays. If the preference does not already exist, `uigetpref` creates it.

Default: None

title

String to display in the dialog box title bar

question

String or cell array of strings specifying a descriptive paragraph for the dialog to display. Use `question` to define what you are asking the user to decide. Clearly state the alternatives and the consequences of choosing among them. The dialog box that `uigetpref` generates inserts line breaks between rows of the string array, between elements of the cell array of strings, and between '`|`' or newline characters within a string vector.

Default: None

pref_choices

String, cell array of strings, or `'|'`-separated strings specifying the labels for the dialog box push buttons. The string on the selected push button is returned.

If the internal preference values are different from the strings displayed on the push buttons, provide a 2-by-n cell array of strings. The first row contains the preference strings, and the second row contains the associated push button strings. When `pref_choices` is not a 2-by-n cell array, MATLAB constructs lowercase versions of the button labels and returns them in `pref_value`. If your code tests returned values, make sure it compares them against the appropriate strings.

Default: None

Name-Value Pair Arguments

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

'CheckboxState'

State of check box when dialog box opens:

- `false` or `0`: Check box is not initially selected.
- `true` or `1`: Check box is initially selected.

Default: `0`

'CheckboxString'

String specifying the label for the check box.

Default: `'Do not show this dialog again'`

'HelpString'

String specifying the label for **Help** button.

Default: No **Help** button displays.

'HelpFcn'

String or function handle specifying the callback that executes when you click the **Help** button. Also, to have a button to trigger the callback, you must specify the `HelpString` option.

Default: `doc('uigetpref')`

'ExtraOptions'

String or cell array of strings specifying extra buttons. The additional buttons are not mapped to any preference settings. If you click any of these buttons, the dialog closes and `uigetpref` returns the string in `value`.

Default: `{}`

'DefaultButton'

String specifying the value that `uigetpref` returns if you close the dialog without clicking a button. This string does not have to correspond to any preference or `ExtraOption`.

Default: The first button specified in `pref_choices`

Output Arguments

pref_value

String containing the preference corresponding to the button you press. If you cancel the dialog by clicking its close box, `uigetpref` returns the label of the first button specified in `pref_choices` or the value for `DefaultButton`, if specified. After you select the **Do not show this dialog again** check box, `uigetpref` does not display a dialog box when you call it with the same `group` and `pref` arguments. Instead, it returns the last choice you made (your registered preference) in `pref_value` immediately.

dlgshown

Logical value that specifies the state of the check box when the dialog box closed. The value is `1` if selected, `0` if not selected. After you select the **Do not show this dialog again** check box, `uigetpref` does not display a dialog box when you call it with the same `group` and `pref` arguments. Instead, it returns `0` in `dlgshown` immediately.

Definitions

Preferences

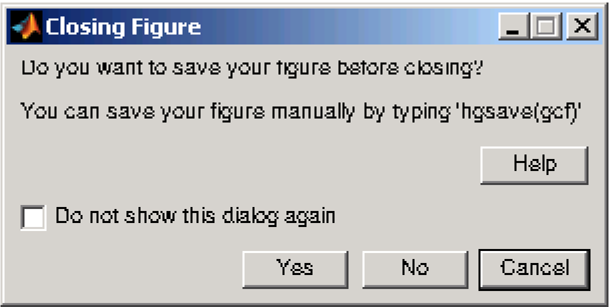
Preferences provide the ability to specify how applications behave and how users interact with them. For example, you can set a preference for which products display their documentation in the Help browser, or which messages the Code Analyzer displays. In MathWorks software products, preferences persist across sessions and are stored in a preference data base, the location of which is system-dependent. Use the **Preferences** option on the **File** menu to access all built-in preferences.

`uigetpref` uses the same preference data base as `addpref`, `getpref`, `ispref`, `rmpref`, and `setpref`. However, `uigetpref` registers the preferences it sets as a separate list, so that it and `uisetpref` can manage those preferences.

To modify preferences registered with `uigetpref`, you can use `setpref` and `uisetpref` to explicitly change preference values to `'ask'`. If you specify a preference that does not already exist in the preference data base, `uigetpref` creates it.

Examples

Create a preference dialog for the `'savefigurebeforeclosing'` preference in the `'mygraphics'` group of preferences.



Call `uigetpref` to display the dialog (or not) from a figure window `CloseRequestFcn` callback. The callback code takes action via a `switch` statement. The action (to delete or not to delete the figure) depends on whether the answer returned by `uigetpref` was `'always'` or `'never'`:

```
function save_figure_perhaps
% Closes figure conditionally, asking whether to save it first.
% User can suppress the dialog from UIGETPREF permanently by selecting
% "Do not show this dialog again".

fig = gcf;
[selectedButton dlgshown] = uigetpref(...
    'mygraphics',...                        % Group
    'savefigurebeforeclosing',...           % Preference
    'Closing Figure',...                    % Window title
    {'Do you want to save your figure before closing?'
     ''
     'You can save your figure manually by typing ''hgsave(gcf)'''},...
    {'always','never';'Yes','No'},...        % Values and button strings
     'ExtraOptions','Cancel',...             % Additional button
     'DefaultButton','Cancel',...            % Default choice
     'HelpString','Help',...                 % String for Help button
     'HelpFcn','doc(''closereq'');');        % Callback for Help button
switch selectedButton
    case 'always'  % Open a Save dialog (without testing if saved before)
    [fileName,pathName,filterIndex] = uiputfile('fig', ...
                              'Save current figure', ...
```

```
                                 'untitled.fig');
        if filterIndex == 0     % User cancelled save or named no file
            delete(fig);
        else                    % Use filename returned from UIPUTFILE
            saveas(fig,[pathName fileName])
            delete(fig);
        end
    case 'never'                % Close the figure without saving it
        delete(fig);
    case 'cancel'               % Do not close the figure
        return
  end
```

To execute the example, copy it and paste the code into a new program file. Name the file `save_figure_perhaps.m` and place it on your search path. To use the function as a `CloseRequestFcn`, create a figure as follows:

```
figure('CloseRequestFcn','save_figure_perhaps');
```

Clicking the figure close box or selecting **File > Close** displays the dialog box until you select **Do not show this dialog again**.

See Also

`ispref` | `addpref` | `getpref` | `setpref` | `prefdir` | `uisetpref` | `rmpref`

Tutorials

- "Preferences"
- "Confirmation Dialogs Preferences"

uiimport

Purpose

Import data interactively

Syntax

```
uiimport
uiimport(filename)
uiimport('-file')
uiimport('-pastespecial')
S = uiimport(___)
```

Description

`uiimport` opens a dialog to interactively load data from a file or the clipboard. MATLAB displays a preview of the data in the file.

`uiimport(filename)` opens the file specified in `filename`.

`uiimport('-file')` presents the file selection dialog first.

`uiimport('-pastespecial')` presents the clipboard contents first.

`S = uiimport(___)` stores the resulting variables as fields in the struct `S`.

See Also

`load` | `importdata` | `clipboard`

How To

- "Supported File Formats"

Purpose

Create menus and menu items on figure windows

Syntax

```
handle = uimenu('PropertyName',PropertyValue,...)
handle = uimenu(parent,'PropertyName',PropertyValue,...)
```

Description

`handle = uimenu('PropertyName',PropertyValue,...)` creates a menu in the current figure's menu bar using the values of the specified properties and assigns the menu handle to `handle`.

`handle = uimenu(parent,'PropertyName',PropertyValue,...)` creates a submenu of a parent menu or a menu item on a context menu specified by `parent` and assigns the menu handle to `handle`. If `parent` refers to a figure instead of another uimenu object or a uicontextmenu, MATLAB software creates a new menu on the referenced figure's menu bar.

Tips

MATLAB adds the new menu to the existing menu bar. If the figure does not have a menu bar, MATLAB creates one. Each menu choice can itself be a menu that displays its items when you select it. `uimenu` accepts property name/property value pairs as well as structures and cell arrays of properties as input arguments.

Use the uimenu `Callback` property to define the action taken when you activate the created menu item.

Uimenus only appear in figures whose `Window Style` is `normal`. If a figure containing uimenu children is changed to `modal`, the uimenu children still exist and are contained in the `Children` list of the figure, but are not displayed until the `WindowStyle` is changed to `normal`.

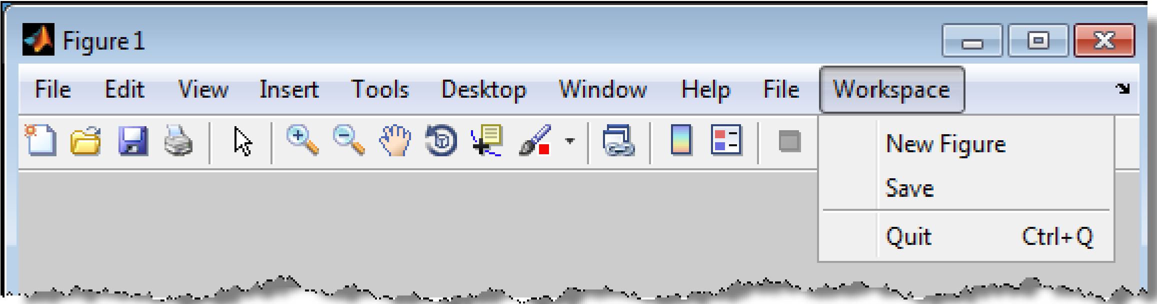
The value of the figure `MenuBar` property affects the content of the figure menu bar. When `MenuBar` is `figure`, a set of built-in menus precedes any user-created uimenus on the menu bar (MATLAB controls the built-in menus and their handles are not available to the user). When `MenuBar` is `none`, uimenus are the only items on the menu bar (that is, the built-in menus do not appear).

See the Uimenu Properties reference page for more information. You can set and query property values after creating a menu with the `set` and `get` functions.

Examples

This example creates a menu labeled **Workspace** with menu options for creating a new figure window, saving workspace variables, and exiting MATLAB. In addition, it defines an accelerator key for the Quit option.

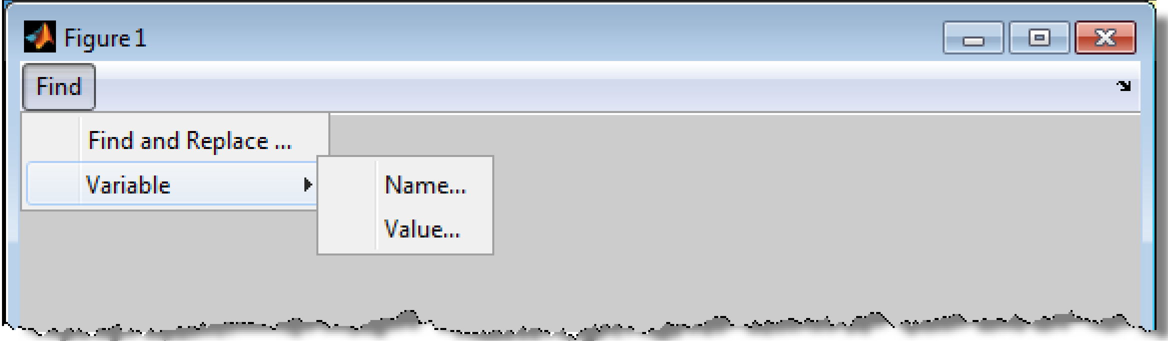
```
f = uimenu('Label','Workspace');
    uimenu(f,'Label','New Figure','Callback','figure');
    uimenu(f,'Label','Save','Callback','save');
    uimenu(f,'Label','Quit','Callback','exit',...
           'Separator','on','Accelerator','Q');
```



This example creates a new figure with a menu bar that excludes the built-in menus. It creates a **Find** menu with options **Find & Replace**, and **Variable**. For the Variable option, it creates a submenu with options of **Name** and **Value**.

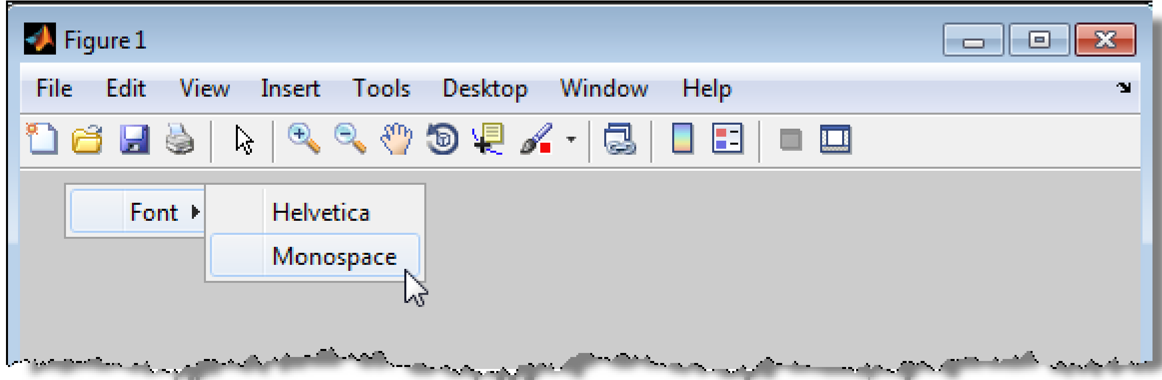
```
f=figure('MenuBar','None');
mh = uimenu(f,'Label','Find');
frh = uimenu(mh,'Label','Find and Replace ...',...
           'Callback','goto');
frh = uimenu(mh,'Label','Variable');
uimenu(frh,'Label','Name...', ...
          'Callback','variable');
```

```
uimenu(frh,'Label','Value...', ...
          'Callback','value');
```



This example creates a context menu, **Font**, on a figure with menu options **Helvetica** and **Monospace**. When you run the code and then right-click anywhere within the figure window, the context menu displays.

```
% Create the UICONTEXTMENU
cmenu = uicontextmenu;

% Create the parent menu
fontmenu = uimenu(cmenu, 'label','Font');

% Create the submenus
font1 = uimenu(fontmenu, 'label','Helvetica','Callback', 'HelvFont');
font2 = uimenu(fontmenu, 'label','Monospace','Callback', 'MonoFont');
set(gcf,'uicontextmenu',cmenu);
```



See Also

Uimenu Properties | `uicontrol` | `uicontextmenu` | `gcbo` | `set` | `get` | `figure`

Purpose

Describe menu properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` commands enable you to set and query the values of properties

You can set default Uimenu properties on the root, figure and menu levels:

```
set(0,'DefaultUimenuPropertyName',PropertyValue...)
set(gcf,'DefaultUimenuPropertyName',PropertyValue...)
set(menu_handle,'DefaultUimenuPropertyName',PropertyValue...)
```

Where *`PropertyName`* is the name of the Uimenu property and `PropertyValue` is the value you specify as the default for that property.

For more information about changing the default value of property see "Setting Default Property Values"

Uimenu Properties

This section lists all properties useful to `uimenu` objects along with valid values and instructions for their use. Curly braces { } enclose default values.

Property Name	Property Description
`Accelerator`	Keyboard equivalent
`BeingDeleted`	This object is being deleted
`BusyAction`	Callback routine interruption
`Callback`	Control action
`Checked`	Menu check indicator
`Children`	Handles of submenus

Property Name	Property Description
`CreateFcn`	Callback routine executed during object creation
`DeleteFcn`	Callback routine executed during object deletion
`Enable`	Enable or disable the uimenu
`ForegroundColor`	Color of text
`HandleVisibility`	Whether handle is accessible from command line and GUIs
`Interruptible`	Callback routine interruption mode
`Label`	Menu label
`Parent`	Uimenu object's parent
`Position`	Relative uimenu position
`Separator`	Separator line mode
`Tag`	User-specified object identifier
`Type`	Class of graphics object
`UserData`	User-specified data
`Visible`	Uimenu visibility

`Accelerator`

character

Keyboard equivalent. An alphabetic character specifying the keyboard equivalent for the menu item. This allows users to select a particular menu choice by pressing the specified character in conjunction with another key, instead of selecting the menu item with the mouse. The key sequence is platform specific:

- For Microsoft Windows systems, the sequence is **Ctrl**+`Accelerator`. Windows reserves these keys for default

menu items: **c**, **v**, and **x**. For more information, see Keyboard shortcuts for Windows on the Microsoft support Web site.

- For Macintosh systems, the sequence is **Cmd**+`Accelerator`. Apple reserves these keys for default menu items: **a**, **c**, **v**, and **x**. For more information, see Mac OS X keyboard shortcuts on the Apple Computer support Web site.
- For UNIX systems, the sequence is **Ctrl**+`Accelerator`. These keys are reserved for default menu items: **o**, **p**, **s**, and **w**. Many UNIX applications also use **a**, **c**, **v**, and **x** in the same manner as the other platforms.

Accelerated menu items do not have to be displayed for the accelerator key to work (you can accelerate a submenu). However, some restrictions apply:

- You can define an accelerator only for menu items that do not have children menus.
- Accelerators work only for menu items that directly execute a callback routine.
- The figure must be in focus when entering the accelerator key sequence.

To remove an accelerator, set `Accelerator` to an empty string, `''`.

For a cross-platform comparison of accelerators, see the article Table of Keyboard Shortcuts.

`BeingDeleted`

`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback routine interruption. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, the callback associated with the new event uses the value of `BusyAction` to decide whether or not to attempt to interrupt the executing callback.

- If the value is `cancel`, the event is discarded and the second callback does not execute.
- If the value is `queue`, and the `Interruptible` property of the first callback is `on`, the second callback is added to the event queue and executes in its turn after the first callback finishes execution.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. See the`Interruptible` property for information about controlling a callback's interruptibility.

`Callback`

string or function handle

Menu action. A callback routine that executes whenever you activate the menu or its submenu (if one exists). Activation is platform dependent. Typically, it is clicking the menu or, if the menu has one, its submenu. Define this routine as a string that

is a valid MATLAB expression or the name of a code file. The expression executes in the MATLAB base workspace.

Caution Do not use a uimenu callback to dynamically change menu items. Deleting, adding, and replacing menu items in a callback can result in a blank menu on some platforms. You can hide, show, and disable menu items in a callback to achieve the same effect. To fully repopulate menu items, delete and create them outside the callback.

`Checked`
`on | {off}`

Menu check indicator. Setting this property to `on` places a check mark next to the corresponding menu item. Setting it to `off` removes the check mark. You can use this feature to create menus that indicate the state of a particular option. For example, suppose you have a menu item called **Show axes** that toggles the visibility of an axes between visible and invisible each time the user selects the menu item. If you want a check to appear next to the menu item when the axes are visible, add the following code to the callback for the **Show axes** menu item:

```
if strcmp(get(gcbo, 'Checked'),'on')
    set(gcbo, 'Checked', 'off');
else
    set(gcbo, 'Checked', 'on');
end
```

This changes the value of the `Checked` property of the menu item from `on` to `off` or vice versa each time a user selects the menu item.

Note that there is no formal mechanism for indicating that an unchecked menu item will become checked when selected.

Note This property is ignored for top level and parent menus.

`Children`

vector of handles

Handles of submenus. A vector containing the handles of all children of the uimenu object. The children objects of uimenus are other uimenus, which function as submenus. You can use this property to reorder the menus.

`CreateFcn`

string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a `uimenu` object. MATLAB sets all property values for the uimenu before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the uimenu being created.

Setting this property on an existing uimenu object has no effect.

You can define a default `CreateFcn` callback for all new uimenus. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uimenu`. For example, the code

```
set(0,'DefaultUimenuCreateFcn','set(gcbo,...
    ''Visible'',''on'')')
```

creates a default `CreateFcn` callback that runs whenever you create a new menu. It sets the default `Visible` property of a `uimenu` object.

To override this default and create a menu whose `Visible` property is set to a different value, call `uimenu` with code similar to

```
hpt = uimenu(...,'CreateFcn','set(gcbo,...
''Visible'',''off'')')
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uimenu` call. In the example above, if instead of redefining the `CreateFcn` property for this uimenu, you had explicitly set `Visible` to `off`, the default `CreateFcn` callback would have set `Visible` back to the default, i.e., `on`.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

See "Function Handle Callbacks" for information on how to use function handles to define a callback function.

`DeleteFcn`

string or function handle

Delete uimenu callback routine. A callback routine that executes when you delete the uimenu object (e.g., when you issue a `delete` command or cause the figure containing the uimenu to reset). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose DeleteFcn is being executed is accessible only through the root `CallbackObject` property, which is more simply queried using gcbo.

See "Function Handle Callbacks" for information on how to use function handles to define a callback function.

`Enable`
`{on} | off`

Enable or disable the uimenu. This property controls whether a menu item can be selected. When not enabled (set to `off`), the menu `Label` appears dimmed, indicating the user cannot select it.

`ForegroundColor`
`ColorSpec` X-Windows only

Color of menu label string. This property determines color of the text defined for the `Label` property. Specify a color using a three-element RGB vector or one of the MATLAB predefined names. The default text color is black. See `ColorSpec` for more information on specifying color.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.

- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to on to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

`Interruptible`
`{on} | off`

Callback routine interruption mode. If a callback is executing and the user triggers an event (such as a mouse click) on an object for which a callback is defined, that callback attempts to interrupt the first callback. MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is executing
- Whether the executing callback contains `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statements
- The `BusyAction` property of the object whose callback is waiting to execute

If the `Interruptible` property of the object whose callback is executing is `on` (the default), the callback can be interrupted. Whenever the callback calls one of the `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` functions, the function processes any events in the event queue, including the waiting callback, before performing its defined task.

If the `Interruptible` property of the object whose callback is executing is `off`, the callback cannot be interrupted (except by certain callbacks; see the note below). The `BusyAction` property

of the object whose callback is waiting to execute determines what happens to the callback.

Note If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. A figure's `WindowButtonDownFcn` callback routine, or an object's `ButtonDownFcn` or `Callback` routine are processed according to the rules described above.

`Label`

string

Menu label. A string specifying the text label on the menu item. You can specify a mnemonic for the label using the `'&'` character. Except as noted below, the character that follows the `'&'` in the string appears underlined and selects the menu item when you type **Alt+** followed by that character while the menu is visible. The `'&'` character is not displayed. To display the `'&'` character in a label, use two `'&'` characters in the string:

'O&pen selection' yields **Open selection**

'Save && Go' yields **Save & Go**

`'Save&&Go'` yields **Save & Go**

`'Save& Go'` yields **Save& Go** (the space is not a mnemonic)

Tip Avoid specifying mnemonics for applications designed to run on Macintosh computers. Accessing menus with mnemonics violates the principles of Apple Human Interface Guidelines. However, you can set the `Accelerator` property to access individual menu items with key sequences on a Mac.

Avoid including the reserved words, `default`, `remove`, and `factory`, in menu labels unless you escape them. They are case sensitive. To use one of these words in the `Label` property, escape it by preceding it with a backslash (`'\'`) character. For example:

`'\remove'` yields **remove**

`'\default'` yields **default**

`'\factory'` yields **factory**

`Parent`
handle

Uimenu's parent. The handle of the uimenu's parent object. The parent of a uimenu object is the figure on whose menu bar it displays, or the uimenu of which it is a submenu. You can move a uimenu object to another figure by setting this property to the handle of the new parent.

`Position`
scalar

Relative menu position. The value of `Position` indicates placement on the menu bar or within a menu. Top-level menus are placed from left to right on the menu bar according to the value of their `Position` property, with 1 representing the left-most position. The individual items within a given menu are placed from top to bottom according to the value of their `Position` property, with 1 representing the top-most position.

`Separator`
: `on | {off}`

Separator line mode. Setting this property to `on` draws a dividing line above the menu item.

`Tag`
: string

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

`Type`
: string (read only)

Class of graphics object. For uimenu objects, `Type` is always the string `'uimenu'`.

`UserData`
: matrix

User-specified data. Any matrix you want to associate with the uimenu object. MATLAB does not use this data, but you can access it using the `set` and `get` commands.

`Visible`
: `{on} | off`

Uimenu visibility. By default, all uimenus are visible. When set to `off`, the uimenu is not visible, but still exists and you can query and set its properties.

Purpose

Convert to 8-bit unsigned integer

Syntax

```
intArray = uint8(array)
```

Description

`intArray = uint8(array)` converts the elements of an array into unsigned 8-bit (1-byte) integers of class `uint8`.

Input Arguments

array

Array of any numeric class, such as `single` or `double`. If `array` is already of class `uint8`, the `uint8` function has no effect.

Output Arguments

intArray

Array of class `uint8`. Values range from 0 to $2^8 - 1$.

The `uint8` function maps any values in `array` that are outside the limit to the nearest endpoint. For example,

```
uint8(2^8)   %  2^8 = 256
```

returns

```
ans =
    255
```

Examples

Convert a `double` array to `uint8`:

```
mydata = uint8(magic(10));
```

Alternatives

When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as `zeros`, `ones` or `eye`), rather than calling an integer conversion function. For example,

```
I = uint8(zeros(100));     % Creates an intermediate array
```

is not as efficient as

```
I = zeros(100, 'uint8');   % Preferred
```

See Also

`double` | `single` | `uint16` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64` | `intmax` | `intmin`

Tutorials

- "Integers"
- "Arithmetic Operations on Integer Classes"

Purpose

Convert to 16-bit unsigned integer

Syntax

```
intArray = uint16(array)
```

Description

`intArray = uint16(array)` converts the elements of an array into unsigned 16-bit (2-byte) integers of class `uint16`.

Input Arguments

array

Array of any numeric class, such as `single` or `double`. If `array` is already of class `uint16`, the `uint16` function has no effect.

Output Arguments

intArray

Array of class `uint16`. Values range from 0 to $2^{16} - 1$.

The `uint16` function maps any values in `array` that are outside the limit to the nearest endpoint. For example,

```
uint16(2^16)   %  2^16 = 65536
```

returns

```
ans =
    65535
```

Examples

Convert a `double` array to `uint16`:

```
mydata = uint16(magic(100));
```

Alternatives

When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as `zeros`, `ones` or `eye`), rather than calling an integer conversion function. For example,

```
I = uint16(zeros(100));     % Creates an intermediate array
```

is not as efficient as

```
I = zeros(100, 'uint16');   % Preferred
```

See Also

`double` | `single` | `uint8` | `uint32` | `uint64` | `int8` | `int16` | `int32` | `int64` | `intmax` | `intmin`

Tutorials

- "Integers"
- "Arithmetic Operations on Integer Classes"

uint32

Purpose

Convert to 32-bit unsigned integer

Syntax

```
intArray = uint32(array)
```

Description

`intArray = uint32(array)` converts the elements of an array into unsigned 32-bit (4-byte) integers of class `uint32`.

Input Arguments

array

Array of any numeric class, such as `single` or `double`. If `array` is already of class `uint32`, the `uint32` function has no effect.

Output Arguments

intArray

Array of class `uint32`. Values range from 0 to $2^{32} - 1$.

The `uint32` function maps any values in `array` that are outside the limit to the nearest endpoint. For example,

```
uint32(2^32)   % 2^32 = 4294967296
```

returns

```
ans =
    4294967295
```

Examples

Convert a `double` array to `uint32`:

```
mydata = uint32(magic(1000));
```

Alternatives

When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as `zeros`, `ones` or `eye`), rather than calling an integer conversion function. For example,

```
I = uint32(zeros(100));     % Creates an intermediate array
```

is not as efficient as

```
I = zeros(100, 'uint32');   % Preferred
```

See Also

`double` | `single` | `uint8` | `uint16` | `uint64` | `int8` | `int16` | `int32` | `int64` | `intmax` | `intmin`

Tutorials

- "Integers"
- "Arithmetic Operations on Integer Classes"

Purpose

Convert to 64-bit unsigned integer

Syntax

```
intArray = uint64(array)
```

Description

`intArray = uint64(array)` converts the elements of an array into unsigned 64-bit (8-byte) integers of class `uint64`.

Tips

Double-precision floating-point numbers have only 52 bits in the mantissa. Therefore, `double` values cannot represent all integers greater than 2^{53} correctly. Before performing arithmetic operations on values larger than 2^{53} in magnitude, convert the values to 64-bit integers. For example,

```
x = uint64(2^53+1);    % Floating-point arithmetic, loses precision
```

is not as accurate as the 64-bit integer arithmetic operation:

```
x = uint64(2^53) + 1;  % Preferred
```

Input Arguments

array

Array of any numeric class, such as `single` or `double`. If `array` is already of class `uint64`, the `uint64` function has no effect.

Output Arguments

intArray

Array of class `uint64`. Values range from 0 to $2^{64} - 1$.

The `uint64` function maps any values in `array` that are outside the limit to the nearest endpoint. For example,

```
uint64(2^64)   % 2^64 = 18446744073709551616
```

returns

```
ans =
    18446744073709551615
```

Examples

Convert a literal value to `uint64`:

```
x = uint64(9007199254740993);
```

Alternatives

When preallocating integer arrays, specify the class in the call to functions that support a class name input (such as `zeros`, `ones` or `eye`), rather than calling an integer conversion function. For example,

```
I = uint64(zeros(100));     % Creates an intermediate array
```

is not as efficient as

```
I = zeros(100, 'uint64');   % Preferred
```

See Also

`double` | `single` | `uint8` | `uint16` | `uint32` | `int8` | `int16` | `int32` | `int64` | `intmax` | `intmin`

Tutorials

- "Integers"
- "Arithmetic Operations on Integer Classes"

Purpose

Interactively select file to open and load data

Syntax

```
uiopen
uiopen(type)
uiopen(filename)
uiopen(filename, TF)
```

Description

`uiopen` displays the dialog box with the file filter set to all MATLAB files (with file extensions `*.m`, `*.mat`, `*.fig`, `*.mdl, and *.slx`).

`uiopen(type)` sets the file filter according to the `type`.

`uiopen(filename)` displays `filename` as the default value for **File name** in the dialog box and lists only files having the same extension.

`uiopen(filename, TF)` directly opens file `filename` without displaying a dialog box if `TF` is true, and displays the dialog box if `TF` is false.

Tips

When you select a file and click **open**, `uiopen` does the following:

- Gets the file using `uigetfile`.
- Opens the file using the `open` command.
  - Files with a file extension of `.m` open in the Editor.
  - Variables stored in files with a file extension of `.mat` appear in the caller's workspace.
  - Files with a file extension of `.fig` open as figure windows.
  - Files with a file extension of `.mdl` or `.slx` open as models in Simulink.
- `uiopen('load')` is the only the form of `uiopen` that you can compile into a standalone application. You can create a file selection dialog box that you can compile using `uigetfile`.
- The `uiopen` dialog box is modal. A modal dialog box prevents you from interacting with other windows until you respond to the modal one.

- `uiopen` displays the same dialog box that opens when you use the MATLAB desktop toolstrip to open a file. (On the **Home** tab, in the **File** section, click **Open** .)

Input Arguments

type

String that specifies the kind of file to show in the dialog box (the file filter). Acceptable values for `type` are the following.

Type string	Files Displayed
`matlab`	All MATLAB files (with file extensions `*.m`, `*.mat`, `*.fig`, `*.mdl`, and `*.slx`.)
`load`	All MAT-files (`*.mat`)
`figure`	All figure files (`*.fig`)
`simulink`	All Simulink model files (`*.mdl` and `*.slx`)
`editor`	All MATLAB files except for `.mat`, `.fig`, and `.slx` files.

filename

A string, including the file extension, naming a file to open. The filename can be a wildcard character plus extension. For example, `*.txt` displays a list of all files with the file extension `.txt`.

TF

A MATLAB expression that evaluates to `true` or `false`. If `true`, `filename` opens directly, without displaying the dialog.

Definitions

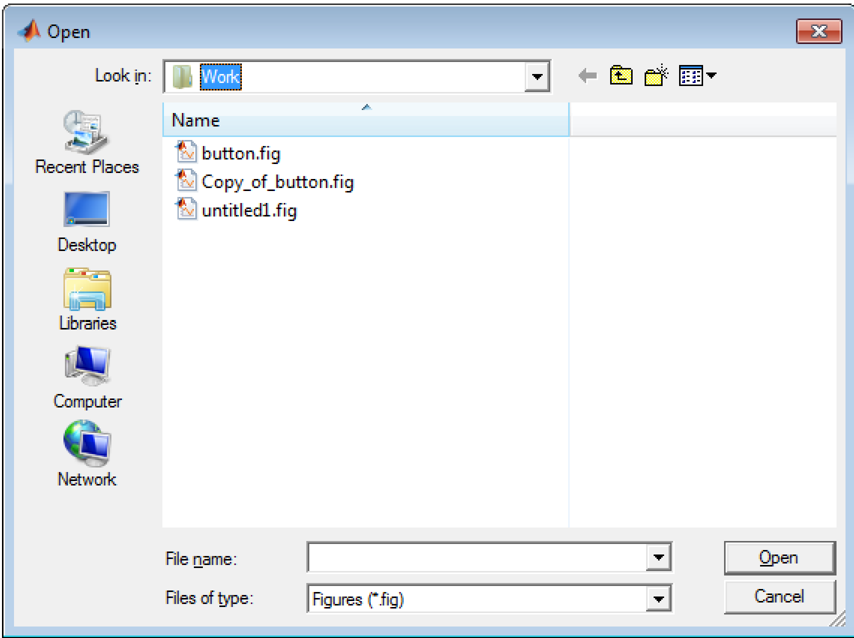
Modal Dialog

A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

Examples

Filter to display only FIG-files by setting the **Files of type** field to `Figures (*.fig)`:

```
uiopen('figure')
```



Alternatives

In MATLAB code or in a command:

- To open a file appropriately based on its file extension, use the `open` function.
- To open a file in the Editor, use the `edit` function.
- To open a MAT-file and load its contents into the workspace, use the `load` function.
- To open a FIG-file, use the `openfig` function.
- To open a file in an application in Microsoft Windows, use the `winopen` function.

See Also

uigetfile | uiputfile | uisave

Purpose

Create panel container object

Syntax

```
h = uipanel('PropertyName1',value1,'PropertyName2',value2,
   ...)
h = uipanel(parent,'PropertyName1',value1,'PropertyName2',
   value2,...)
```

Description

A uipanel groups components. It can contain user interface controls with which the user interacts directly. It can also contain axes, other uipanels, and uibuttongroups. It cannot contain ActiveX controls.

`h = uipanel('PropertyName1',value1,'PropertyName2',value2,...)` creates a `uipanel` container object in a figure, uipanel, or uibuttongroup. Use the `Parent` property to specify the parent figure, uipanel, or uibuttongroup. If you do not specify a parent, `uipanel` adds the panel to the current figure. If no figure exists, one is created. See the Uipanel Properties reference page for more information.

`h = uipanel(parent,'PropertyName1',value1,'PropertyName2',value2,...)` creates a uipanel in the object specified by the handle, `parent`. If you also specify a different value for the `Parent` property, the value of the `Parent` property takes precedence. `parent` must be a `figure`, `uipanel`, or `uibuttongroup`.

A `uipanel` object can have `axes`, `uicontrol`, `uipanel`, and `uibuttongroup` objects as children. For the children of a uipanel, the `Position` property is interpreted relative to the uipanel. If you move the panel, the children automatically move with it and maintain their positions relative to the panel.

After creating a `uipanel` object, you can set and query its property values using `set` and `get`.

Tips

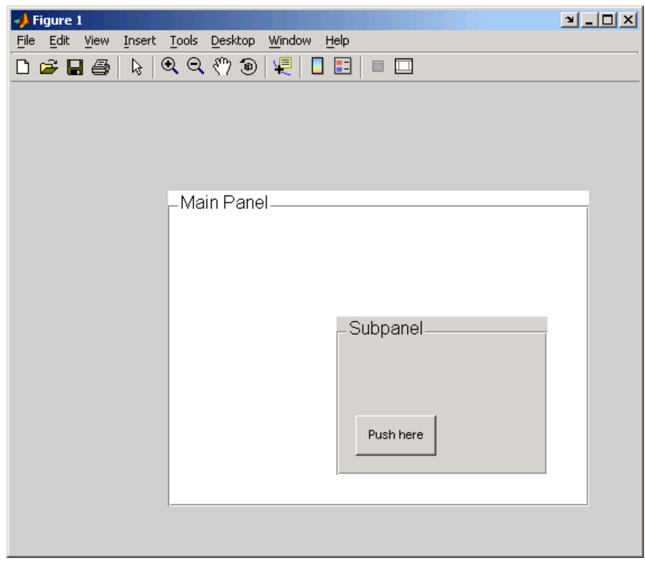
If you set the `Visible` property of a uipanel object to `'off'`, any child objects it contains (buttons, button groups, axes, etc.) become invisible along with the panel itself. However, doing this does *not* affect the

settings of the `Visible` property of any of its child objects, even though all of them remain invisible until the uipanel's visibility is set to `'on'`. `uibuttongroup` components also behave in this manner.

Examples

This example creates a uipanel in a figure, then creates a subpanel in the first panel. Finally, it adds a pushbutton to the subpanel. Both panels use the default `Units` property value, `normalized`. Note that default `Units` for the `uicontrol` pushbutton is pixels.

```
h = figure;
hp = uipanel('Title','Main Panel','FontSize',12,...
             'BackgroundColor','white',...
             'Position',[.25 .1 .67 .67]);
hsp = uipanel('Parent',hp,'Title','Subpanel','FontSize',12,...
              'Position',[.4 .1 .5 .5]);
hbsp = uicontrol('Parent',hsp,'String','Push here',...
              'Position',[18 18 72 36]);
```



See Also

`hgtransform` | `uibuttongroup` | `uicontrol`

Uipanel Properties

Purpose

Describe panel properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

You can set default uipanel properties by typing:

```
set(h,'DefaultUipanelPropertyName',PropertyValue...)
```

Where h can be the root handle (0), a figure handle, or a uipanel handle. *`PropertyName`* is the name of the uipanel property and `PropertyValue` is the value you specify as the default for that property.

Note Default properties you set for uipanels also apply to uibuttongroups.

For more information about changing the default value of a property see "Setting Default Property Values". For an example, see the `CreateFcn` property.

Uipanel Properties

This section lists all properties useful to `uipanel` objects along with valid values and a descriptions of their use. Curly braces { } enclose default values.

Property Name	Description
`BackgroundColor`	Color of the uipanel background
`BeingDeleted`	This object is being deleted
`BorderType`	Type of border around the uipanel area.

Property Name	Description
BorderWidth	Width of the panel border.
BusyAction	Interruption of other callback routines
ButtonDownFcn	Button-press callback routine
Children	All children of the uipanel
Clipping	Clipping of child axes, uipanels, and uibuttongroups to the uipanel. Does not affect child uicontrols.
CreateFcn	Callback routine executed during object creation
DeleteFcn	Callback routine executed during object deletion
FontAngle	Title font angle
FontName	Title font name
FontSize	Title font size
FontUnits	Title font units
FontWeight	Title font weight
ForegroundColor	Title font color and/or color of 2-D border line
HandleVisibility	Handle accessibility from commandline and GUIs
HighlightColor	3-D frame highlight color
HitTest	Selectable by mouse click
Interruptible	Callback routine interruption mode
Parent	Uipanel object's parent
Position	Panel position relative to parent figure or uipanel
ResizeFcn	User-specified resize routine

Property Name	Description
`Selected`	Whether object is selected
`SelectionHighlight`	Object highlighted when selected
`ShadowColor`	3-D frame shadow color
`Tag`	User-specified object identifier
`Title`	Title string
`TitlePosition`	Location of title string in relation to the panel
`Type`	Object class
`UIContextMenu`	Associates uicontextmenu with the uipanel
`Units`	Units used to interpret the position vector
`UserData`	User-specified data
`Visible`	Uipanel visibility. **Note** Controls the visibility of a uipanel and of its child axes, uibuttongroups. uipanels, and child uicontrols. Setting it does not change their `Visible` property.

`BackgroundColor`
`ColorSpec`

Color of the uipanel background. A three-element RGB vector or one of the MATLAB predefined names, specifying the background color. See the `ColorSpec` reference page for more information on specifying color.

`BeingDeleted`
`on | {off}` (read-only)

This object is being deleted. Mechanism to determine if objects are in the process of being deleted. MATLAB sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, an object's delete function calls other functions that act on a number of different objects. If a function does not need to perform an action on an about-be-deleted object, it can check the object's `BeingDeleted` property before acting.

`BorderType`

`none | {etchedin} | etchedout | beveledin | beveledout | line`

Border of the uipanel area. Used to define the panel area graphically. Etched and beveled borders provide a 3-D look. Use the `HighlightColor` and `ShadowColor` properties to specify the border color of etched and beveled borders. A line border is 2-D. Use the `ForegroundColor` property to specify its color.

`BorderWidth`

`integer`

Width of the panel border. The width of the panel borders in pixels. The default border width is 1 pixel. 3-D borders wider than 3 may not appear correctly at the corners.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the

running callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`
string or function handle

Button-press callback routine. A callback routine that executes when you press a mouse button while the pointer is in a 5-pixel wide border around the uipanel. This is useful for implementing actions to interactively modify control object properties, such as size and position, when they are clicked on (using the selectmoveresize function, for example).

If you define this routine as a string, the string can be a valid MATLAB expression or the name of a code file. The expression executes in the MATLAB workspace.

`Children`
vector of handles

Children of the uipanel. A vector containing the handles of all children of the uipanel. A `uipanel` object's children are axes, uipanels, uibuttongroups, and uicontrols. You can use this property to reorder the children.

`Clipping`
`{on} | off`

Clipping mode. By default, MATLAB clips a uipanel's child axes, uipanels, and uibuttongroups to the uipanel rectangle. If you set `Clipping` to `off`, the axis, uipanel, or uibuttongroup is displayed outside the panel rectangle. This property does not affect child uicontrols which, by default, can display outside the panel rectangle.

`CreateFcn`

string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a `uipanel` object. MATLAB sets all property values for the uipanel before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the uipanel being created.

Setting this property on an existing uipanel object has no effect.

You can define a default `CreateFcn` callback for all new uipanels. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uipanel`. For example, the code

```
set(0,'DefaultUipanelCreateFcn','set(gcbo,...
    ''FontName'',''arial'',''FontSize'',12)')
```

creates a default `CreateFcn` callback that runs whenever you create a new panel. It sets the default font name and font size of the uipanel title.

Note Uibuttongroup takes its default property values from uipanel. Defining a default property for all uipanels defines the same default property for all uibuttongroups.

To override this default and create a panel whose `FontName and FontSize` properties are set to different values, call `uipanel` with code similar to

```
hpt = uipanel(...,'CreateFcn','set(gcbo,...
''FontName'',''times'',''FontSize'',14)')
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uipushtool` call. In the example above, if instead of redefining the `CreateFcn` property for this uipanel, you had explicitly set `Fontsize` to `14`, the default `CreateFcn` callback would have set `FontSize` back to the system dependent default.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

See "Function Handle Callbacks" for information on how to use function handles to define a callback function.

`DeleteFcn`
string or function handle

Callback routine executed during object deletion. A callback routine that executes when you delete the uipanel object (e.g., when you issue a delete command or clear the figure containing the uipanel). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine. The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

`FontAngle`
    `{normal} | italic | oblique`

Character slant used in the Title. MATLAB uses this property to select a font from those available on your particular system. Setting this property to `italic` or `oblique` selects a slanted version of the font, when it is available on your system.

`FontName`
    string

Font family used in the Title. The name of the font in which to display the `Title`. To display and print properly, this must be a font that your system supports. The default font is system dependent. To eliminate the need to hard code the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan), set `FontName` to the string `FixedWidth` (this string value is case insensitive).

```
set(uicontrol_handle,'FontName','FixedWidth')
```

This then uses the value of the root `FixedWidthFontName` property which can be set to the appropriate value for a locale from `startup.m` in the end user's environment. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font

`FontSize`
    integer

Title font size. A number specifying the size of the font in which to display the Title, in units determined by the `FontUnits` property. The default size is system dependent.

`FontUnits`
    `inches | centimeters | normalized | {points} |pixels`

Title font size units. `Normalized` units interpret `FontSize` as a fraction of the height of the uipanel. When you resize the uipanel, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).

`FontWeight`
: `light | {normal} | demi | bold`

Weight of characters in the title. MATLAB uses this property to select a font from those available on your particular system. Setting this property to bold causes MATLAB to use a bold version of the font, when it is available on your system.

`ForegroundColor`
: `ColorSpec`

Color used for title font and 2-D border line. A three-element RGB vector or one of the MATLAB predefined names, specifying the font or line color. See the `ColorSpec` reference page for more information on specifying color.

`HandleVisibility`
: `{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.

- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to on to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

`HighlightColor`
`ColorSpec`

3-D frame highlight color. A three-element RGB vector or one of the MATLAB predefined names, specifying the highlight color. See the `ColorSpec` reference page for more information on specifying color.

`HitTest`
`{on} | off`

Selectable by mouse click. `HitTest` determines if the uipanel can become the current object (as returned by the `gco` command and the figure `CurrentObject` property) as a result of a mouse click on the panel. If `HitTest` is `off`, clicking the panel sets the `CurrentObject` to the closest ancestor of the panel that registers `HitTest`. The uipanel property `HandleVisibility` must be `'on'` for it to become the `CurrentObject`. If the uipanel `HandleVisibility` is `'off'` or `'callback'`, or if the panel and all

its ancestors have `HitTest` set to `'off'`, the figure `CurrentObject` is the empty matrix.

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For user interface objects, the `Interruptible` property affects the callbacks for these properties only:

- `ButtonDownFcn`
- `KeyPressFcn`
- `KeyReleaseFcn`
- `WindowButtonDownFcn`
- `WindowButtonMotionFcn`
- `WindowButtonUpFcn`
- `WindowKeyPressFcn`
- `WindowKeyReleaseFcn`
- `WindowScrollWheelFcn`

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:

- If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
- If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

The `BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting the `Interruptible` property to `on` (default), allows a callback from other user interface objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see “Control Callback Execution and Interruption”.

Note If the interrupting callback is a `DeleteFcn`or `CreateFcn` callback, or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. An object's `ButtonDownFcn` or `Callback` routine is processed according to the rules described previously in this section.

`Parent`
handle

Uipanel parent. The handle of the uipanel's parent figure, uipanel, or uibuttongroup. You can move a uipanel object to another figure, uipanel, or uibuttongroup by setting this property to the handle of the new parent.

`Position`
position rectangle

Size and location of uipanel relative to parent. The rectangle defined by this property specifies the size and location of the panel within the parent figure window, uipanel, or uibuttongroup. Specify `Position` as

```
[left bottom width height]
```

`left` and `bottom` are the distance from the lower-left corner of the parent object to the lower-left corner of the `uipanel` object. `width` and `height` are the dimensions of the uipanel rectangle, including the title. All measurements are in units specified by the `Units` property.

`ResizeFcn`
function handle or string

Resize callback routine. MATLAB executes this callback routine whenever a user manually or programmatically resizes the uipanel, or in GUIDE, the Resize behavior option is set to `other`. You can query the uipanel `Position` property to determine its new size and position. During execution of the callback routine, the handle to the uipanel being resized is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

All axes, uipanel, uitable and uicontrol objects that have their `Units` set to `normalized` automatically resize proportionally to the figure. You can define individual resize functions for any such object as needed. For example, you can use `ResizeFcn` to maintain a GUI layout that is not directly supported by the MATLAB `Position`-and-`Units` paradigm.

For example, consider a GUI layout that maintains an object attached to the top of the figure at a constant height in pixels, but always matches the width of the figure. The following `ResizeFcn` accomplishes this; it keeps the uicontrol whose `Tag` is `'StatusBar'` 20 pixels high, as wide as the figure, and attached to the top of the figure. Note the use of the `Tag` property to retrieve the uicontrol handle, and the `gcbo` function to retrieve the figure handle. The code also does “defensive programming” to save and restores figure `Units`. The callback requires `Units` in pixels in order to work correctly, but takes care not to permanently alter that property.

```
u = findobj('Tag','StatusBar');
fig = gcbo;
old_units = get(fig,'Units');
set(fig,'Units','pixels');
figpos = get(fig,'Position');
upos = [0, figpos(4) - 20, figpos(3), 20];
set(u,'Position',upos);
set(fig,'Units',old_units);
```

You can change the figure `Position` from within a uipanel `ResizeFcn` callback; however, the `ResizeFcn` is not called again as a result.

A figure's uipanels resize before the figure itself does. Nested uipanels resize from inner to outer, with child `ResizeFcn`s being called before parent `ResizeFcn`s.

The `print` command can cause the `ResizeFcn` to be called if the `PaperPositionMode` property is set to `manual` and you have defined a resize function. If you do not want your resize function called by `print`, set the `PaperPositionMode` to `auto`.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

See Resize Behavior for information on creating resize functions using GUIDE.

`Selected`
`on | off` (read only)

Is object selected? This property indicates whether the panel is selected. When this property is `on`, MATLAB displays selection handles if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`
`{on} | off`

Object highlighted when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`ShadowColor`
`ColorSpec`

3-D frame shadow color. A three-element RGB vector or one of the MATLAB predefined names, specifying the shadow color. See the `ColorSpec` reference page for more information on specifying color.

`Tag`
string

User-specified object identifier. The `Tag` property provides a means to identify graphics objects with a user-specified label. You can define `Tag` as any string.

With the `findobj` function, you can locate an object with a given `Tag` property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified figures) that have the `Tag` value `'FormatTb'`.

```
h = findobj(figurehandles,'Tag','FormatTb')
```

`Title`
string

Title string. The text displayed in the panel title. Vertical slash ('`|`') characters are not interpreted as line breaks and instead show up in the text displayed in the uipanel title. You can position the title using the `TitlePosition` property.

Setting a property value to `default`, `remove`, or `factory` produces the effect described in "Defining Default Values". To set `Title` to one of these words, you must precede the word with the backslash character. For example,

```
hp = uipanel(...,'Title','\Default');
```

`TitlePosition`
`{lefttop} | centertop | righttop | leftbottom | centerbottom | rightbottom`

Location of the title. This property determines the location of the title string, in relation to the uipanel.

`Type`
: `string` (read-only)

  Object class. This property identifies the kind of graphics object. For uipanel objects, `Type` is always the string `'uipanel'`.

`UIContextMenu`
: handle

  Associate a context menu with a uipanel. Assign this property the handle of a `Uicontextmenu` object. MATLAB displays the context menu whenever you right-click the `uipanel`. Use the `uicontextmenu` function to create the context menu.

`Units`
: `inches | centimeters | {normalized} | points | pixels | characters`

  Units of measurement. MATLAB uses these units to interpret the `Position` property. For the panel itself, units are measured from the lower-left corner of the figure window. For children of the panel, they are measured from the lower-left corner of the panel.

  - `Normalized` units map the lower-left corner of the panel or figure window to (0,0) and the upper-right corner to (1.0,1.0).
  - `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).
  - `Character` units are characters using the default system font; the width of one character is the width of the letter x, the height of one character is the distance between the baselines of two lines of text.

  If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not

to affect other functions that assume `Units` is set to the default value.

`UserData`

matrix

User-specified data. Any data you want to associate with the `uipanel` object. MATLAB does not use this data, but you can access it using `set` and `get`.

`Visible`

`{on} | off`

Uipanel visibility. By default, a `uipanel` object is visible. When set to `'off'`, the uipanel is not visible, as are all child objects of the panel. When a panel is hidden in this manner, you can still query and set its properties.

Note The value of a uipanel's `Visible` property determines whether its child components, such as axes, buttons, uibuttongroups, and other uipanels, are visible. However, changing the `Visible` property of a panel does *not* change the settings of the `Visible` property of its child components even though hiding the panel causes them to be hidden.

Purpose

Create push button on toolbar

Syntax

```
hpt = uipushtool
hpt = uipushtool('PropertyName1',value1,'PropertyName2',
   value2,...)
hpt = uipushtool(ht,...)
```

Description

`hpt = uipushtool` creates a push button on the uitoolbar at the top of the current figure window, sets all its properties to default values, and returns a handle to the tool. If no uitoolbar exists, one is created. The uitoolbar is the parent of the uipushtool. Use the returned handle `hpt` to set properties of the tool. The `ClickedCallback` passes the handle as its first argument. The button has no icon, but its border highlights when you hover over it with the mouse cursor. Add an icon by setting `CData` for the tool.

`hpt = uipushtool('`*`PropertyName1`*`',value1,'`*`PropertyName2`*`',value2,...)`, creates a uipushtool and returns a handle to it. `uipushtool` assigns the specified property values, and assigns default values to the remaining properties. You can change the property values at a later time using the `set` function. You can specify properties as parameter name/value pairs, cell arrays containing parameter names and values, or structures with fields containing parameter names and values as input arguments. For a complete list, see Uipushtool Properties. Type `get(hpt)` to see a list of uipushtool object properties and their current values. Type `set(hpt)` to see a list of uipushtool object properties that you can set and their legal property values.

`hpt = uipushtool(ht,...)` creates a button with `ht` as a parent. `ht` must be a uitoolbar handle.

Uipushtools appear in figures whose `Window Style` is `'normal'` or `'docked'`. Push tools do not appear in figures with `'modal'` `WindowStyle`. If you change the `WindowStyle` of a figure containing a uitoolbar and its uipushtool children to `'modal'`, the uipushtools continue to exist as `Children` of the uitoolbar. However, they do

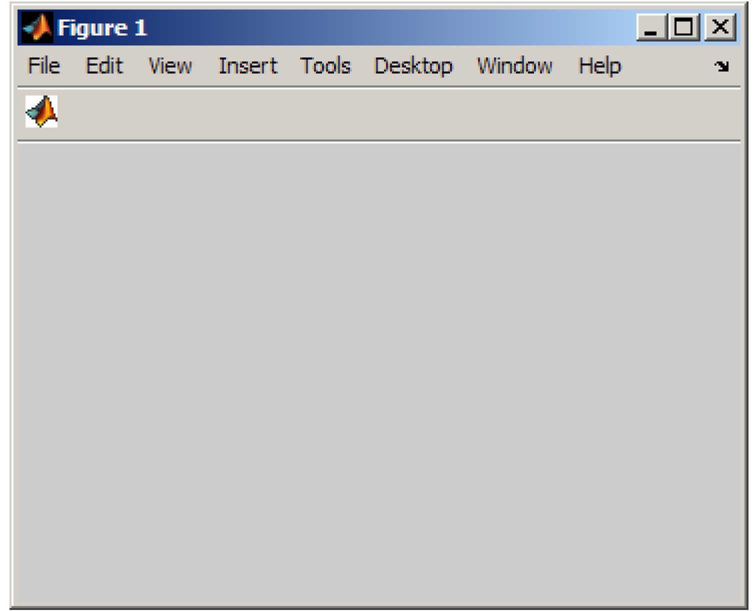
not display until you change the figure `WindowStyle` to `'normal'` or `'docked'`.

Unlike push buttons, uipushtools have no way to indicate that you have double-clicked them. That is, a double click does not set the figure `SelectionType` property to `'open'`. Double-clicking a uipushtool simply executes its `ClickedCallback` twice in succession. Also, uipushtools cannot have context menus.

Examples

Create a uitoolbar object and places a uipushtool object on it. Generate an icon for the tool by reading a GIF file containing a MATLAB icon. Convert the indexed image to a truecolor image before specifying it as `CData`.

```
h  = figure('ToolBar','none');
ht = uitoolbar(h);
% Use a MATLAB icon for the tool
[X map] = imread(fullfile(...
    matlabroot,'toolbox','matlab','icons','matlabicon.gif'));
% Convert indexed image and colormap to truecolor
icon = ind2rgb(X,map);
% Create a uipushtool in the toolbar
hpt = uipushtool(ht,'CData',icon,...
                 'TooltipString','Toolbar push button',...
                 'ClickedCallback',...
                 'disp(''Thank you for clicking a uipushtool.'')')
```



Alternatives

You can also create toolbars with push tools using GUIDE.

See Also

`get` | `set` | `uicontrol` | `uitoggletool` | `uitoolbar` | Uipushtool Properties

Tutorials

- "GUI with Axes, Menu, and Toolbar"

How To

- "Create Toolbars for Programmatic GUIs"

Purpose

Describe push tool properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

You can set default Uipushtool properties by typing:

```
set(h,'DefaultUipushtoolPropertyName',PropertyValue...)
```

Where h can be the root handle (0), a figure handle, a uitoolbar handle, or a uipushtool handle. *`PropertyName`* is the name of the `Uipushtool` property and `PropertyValue` is the value you specify as the default for that property.

For more information about changing the default value of a property see Setting Default Property Values.

Uipushtool Properties

This section lists all properties useful to `uipushtool` objects along with valid values and a descriptions of their use. Curly braces { } enclose default values.

Property	Purpose
`BeingDeleted`	This object is being deleted.
`BusyAction`	Callback routine interruption.
`CData`	Truecolor image displayed on the control.
`ClickedCallback`	Control action.
`CreateFcn`	Callback routine executed during object creation.
`DeleteFcn`	Delete uipushtool callback routine.

Property	Purpose
Enable	Enable or disable the uipushtool.
HandleVisibility	Control access to object's handle.
HitTest	Whether selectable by mouse click
Interruptible	Callback routine interruption mode.
Parent	Handle of uipushtool's parent.
Separator	Separator line mode
Tag	User-specified object label.
TooltipString	Content of object's tooltip.
Type	Object class.
UIContextMenu	Uicontextmenu object associated with the uipushtool
UserData	User specified data.
Visible	Uipushtool visibility.

`BeingDeleted`

`on | {off}` (read only)

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB software sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property). It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, some functions may not need to perform actions on objects that are being deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`CData`

3-dimensional array

Truecolor image displayed on control. An *n*-by-*m*-by-3 array of RGB values that defines a truecolor image displayed on either a push button or toggle button. Each value must be between 0.0 and 1.0. If your `CData` array is larger than 16 in the first or second dimension, it may be clipped or cause other undesirable effects. If the array is clipped, only the center 16-by-16 part of the array is used.

`ClickedCallback`

string or function handle

Control action. A routine that executes when the uipushtool's `Enable` property is set to `on`, and you press a mouse button while the pointer is on the push tool itself or in a 5-pixel wide border around it.

`CreateFcn`

string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a uipushtool object. MATLAB sets all property values for the uipushtool before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the push tool being created.

Setting this property on an existing uipushtool object has no effect.

You can define a default `CreateFcn` callback for all new uipushtools. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uipushtool`. For example, the code

```
imga(:,:,1) = rand(20);
imga(:,:,2) = rand(20);
imga(:,:,3) = rand(20);
set(0,'DefaultUipushtoolCreateFcn','set(gcbo,''Cdata'',imga)')
```

creates a default `CreateFcn` callback that runs whenever you create a new push tool. It sets the default image `imga` on the push tool.

To override this default and create a push tool whose `Cdata` property is set to a different image, call `uipushtool` with code similar to

```
a = [.05:.05:0.95];
imgb(:,:,1) = repmat(a,19,1)';
imgb(:,:,2) = repmat(a,19,1);
```

```
imgb(:,:,3) = repmat(flipdim(a,2),19,1);
hpt = uipushtool(...,'CreateFcn','set(gcbo,''CData'',imgb)',...)
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uipushtool` call. In the example above, if instead of redefining the `CreateFcn` property for this push tool, you had explicitly set `CData` to `imgb`, the default `CreateFcn` callback would have set `CData` back to `imga`.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

See "Function Handle Callbacks" for information on how to use function handles to define a callback function.

`DeleteFcn`

string or function handle

Callback routine executed during object deletion. A callback routine that executes when you delete the uipushtool object (e.g., when you call the `delete` function or cause the figure containing the uipushtool to reset). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose DeleteFcn is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define a callback function.

`Enable`
`{on} | off`

Enable or disable the uipushtool. This property controls how uipushtools respond to mouse button clicks, including which callback routines execute.

- `on` – The uipushtool is operational (the default).
- `off` – The uipushtool is not operational and its image (set by the `Cdata` property) is grayed out.

When you left-click a uipushtool whose `Enable` property is `on`, MATLAB performs these actions in this order:

1. Executes the push tool's `ClickedCallback` routine.
2. Does *not* set the figure `CurrentPoint` property and does not execute the figure's `WindowButtonDownFcn` callback.
3. Does *not* set the figure `SelectionType` property.

When you left-click a uipushtool whose `Enable` property is `off`, or when you right-click a uipushtool whose `Enable` property has any value, no action is reported, no callback executes, and neither the `SelectionType` nor `CurrentPoint` figure properties are modified.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to on to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

`HitTest`
`{on} | off`

Selectable by mouse click. This property has no effect on uipushtool objects.

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For user interface objects, the `Interruptible` property affects the callbacks for these properties only:

- `ButtonDownFcn`
- `KeyPressFcn`

- `KeyReleaseFcn`
- `WindowButtonDownFcn`
- `WindowButtonMotionFcn`
- `WindowButtonUpFcn`
- `WindowKeyPressFcn`
- `WindowKeyReleaseFcn`
- `WindowScrollWheelFcn`

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

The `BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting the `Interruptible` property to `on` (default), allows a callback from other user interface objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see "Control Callback Execution and Interruption".

Note If the interrupting callback is a `DeleteFcn`nor `CreateFcn` callback, or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. An object's `ButtonDownFcn` or `Callback` routine is processed according to the rules described previously in this section.

`Parent`

handle

Uipushtool parent. The handle of the uipushtool's parent toolbar. You can move a uipushtool object to another toolbar by setting this property to the handle of the new parent.

`Separator`

`on | {off}`

Separator line mode. Setting this property to `on` draws a dividing line to the left of the uipushtool.

`Tag`

string

User-specified object identifier. The `Tag` property provides a means to identify graphics objects with a user-specified label. You can define `Tag` as any string.

With the `findobj` function, you can locate an object with a given `Tag` property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified toolbars) that have the `Tag` value `'Copy'`.

```
h = findobj(uitoolbarhandles,'Tag','Copy')
```

`TooltipString`
string

Content of tooltip for object. The `TooltipString` property specifies the text of the tooltip associated with the uipushtool. When the user moves the mouse pointer over the control and leaves it there, the tooltip is displayed.

To create a tooltip that has more than one line of text, use `sprintf` to generate a string containing newline (`\n`) characters and then set the `TooltipString` to that value. For example:

```
h = uipushtool;
s = sprintf('Pushtool tooltip line 1\nPushtool tooltip line 2');
set(h,'TooltipString',s)
```

`Type`
string (read-only)

Object class. This property identifies the kind of graphics object. For uipushtool objects, `Type` is always the string `'uipushtool'`.

`UIContextMenu`
handle

Associate a context menu with uicontrol. This property has no effect on uipushtool objects.

`UserData`

array

User specified data. You can specify `UserData` as any array you want to associate with the uipushtool object. The object does not use this data, but you can access it using the `set` and `get` functions.

`Visible`

`{on} | off`

Uipushtool visibility. By default, all uipushtools are visible. When set to `off`, the uipushtool is not visible, but still exists and you can query and set its properties.

uiputfile

Purpose

Open standard dialog box for saving files

Syntax

```
FileName = uiputfile
[FileName,PathName] = uiputfile
[FileName,PathName,FilterIndex] = uiputfile(FilterSpec)
[FileName,PathName,FilterIndex] = uiputfile(FilterSpec,
   DialogTitle)
[FileName,PathName,FilterIndex] = uiputfile(FilterSpec,
   DialogTitle,DefaultName)
```

Description

Note Successful execution of `uiputfile` does not create a file; it only returns the name of a new or existing file that you designate.

`FileName = uiputfile` displays a modal dialog box for selecting or specifying a file you want to create or save. The dialog box lists the files and folders in the current folder. If the selected or specified file name is valid, `uiputfile` returns it in `FileName`.

`[FileName,PathName] = uiputfile` also returns the path to `FileName` in `PathName`, or if you cancel the dialog, returns `0` for both arguments. If you do not provide any output arguments, the file name alone is returned in `ans`.

`[FileName,PathName,FilterIndex] = uiputfile(FilterSpec)` displays only those files with extensions that match `FilterSpec`. On some platforms `uiputfile` also displays in gray any files that do not match `FilterSpec`. The `uiputfile` function appends `'All Files'` to the list of file types. `FilterSpec` can be a string or a cell array of strings, and can include the `*` and `?` wildcard characters. For example, `'*.m'` lists all MATLAB program files in a folder.

`FilterSpec` can be a string that contains a file name. `uiputfile` displays the file name selected in the **File name** field and uses the file extension as the default filter. The `FilterSpec` string can include a path, or consist of a path only. To specify a folder only, make the

last character in `DefaultName` `'\'` or `'/'`. A path can contain special path characters, such as `'.'`, `'..'`, `'/'`, `'\'`, or `'~'`. For example, `'../*.m'` lists all program files in the folder above the current folder. If `FilterSpec` is a cell array of strings, the first column contains a list of file extensions. The optional second column contains a corresponding list of descriptions. These descriptions replace the default descriptions in the **Save as type** pop-up menu. A description cannot be an empty string. See the "Examples" on page 1-5853 for illustration of using cell arrays as `FilterSpec`. If you do not specify `FilterSpec`, `uiputfile` uses the default list of file types (all MATLAB files). `FilterIndex` is the index of the filter selected in the dialog box. Indexing starts at 1. If you click the **Cancel** button, close the dialog window, or if the file does not exist, `uiputfile` returns `FilterIndex` as `0`.

`[FileName,PathName,FilterIndex] = uiputfile(FilterSpec,DialogTitle)` displays a dialog box that has the title `DialogTitle`. To use the default file types and to specify a dialog title, enter `uiputfile('','DialogTitle')`

`[FileName,PathName,FilterIndex] = uiputfile(FilterSpec,DialogTitle,DefaultName)` displays a dialog box in which the file name specified by `DefaultName` appears in the **File name** field. `DefaultName` can also be a path or a path+filename. To specify a folder only, make the last character in `DefaultName` `'\'` or `'/'`. In this case, `uiputfile` opens the dialog box in the folder specified by the path. If you specify a path in `DefaultName` that does not exist, `uiputfile` opens the dialog box in the current folder. You can use `'.'`,`'..'`, `'\'`, `'/'`, or ~ in the `DefaultName` argument.

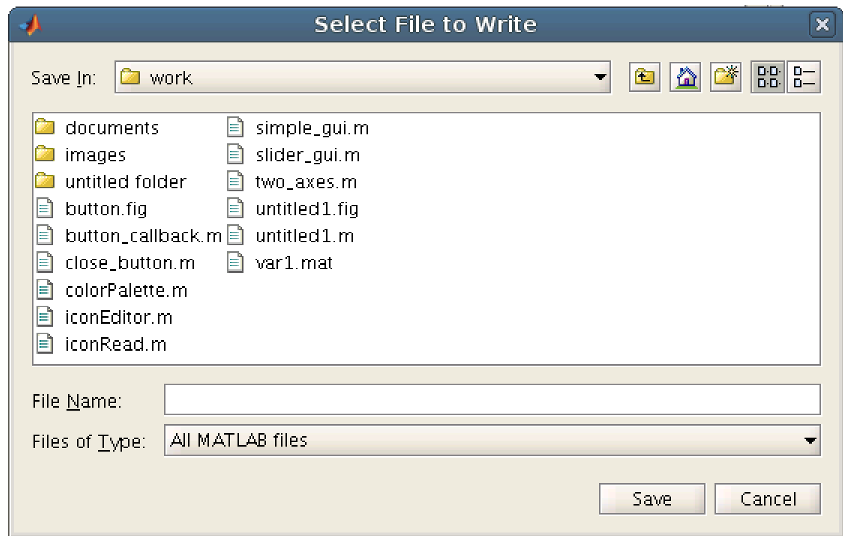
When typing into the dialog box, if you include either of the wildcard characters `'*'` or `'?'` in a file name, `uiputfile` does not respond to clicking **Save**. The dialog box remains open until you cancel it or remove the wildcard characters. This restriction applies to all platforms, even to file systems that permit these characters in file names.

If you select or specify an existing file name, a warning dialog box opens stating that the file already exists and asks if you want to replace it.

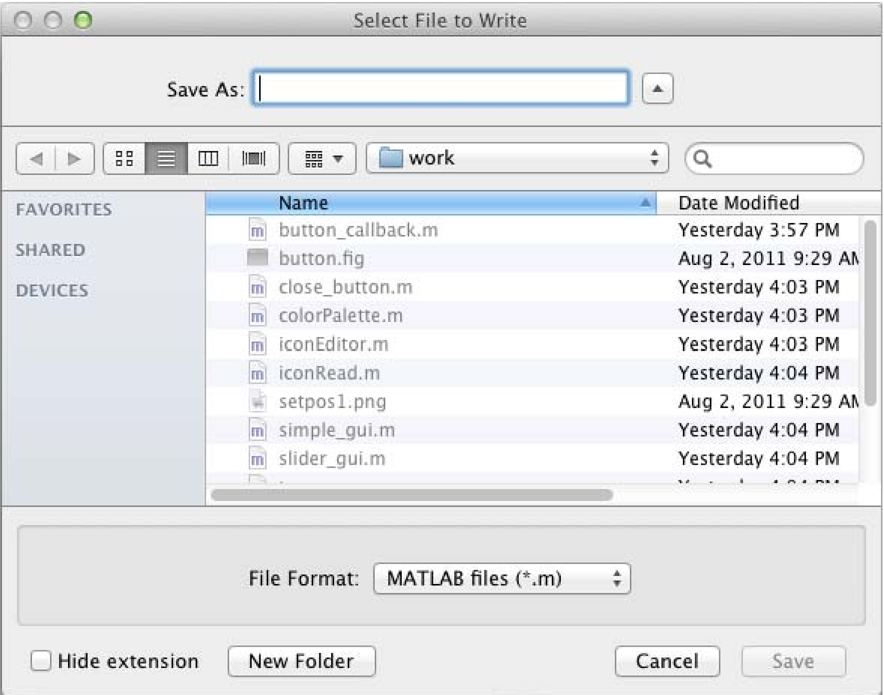
Select **Yes** to replace the existing file or **No** to return to the dialog to select another file name. Selecting **Yes** returns the name of the file. Selecting **No** returns 0.

For Microsoft Windows platforms, the dialog box is the Windows dialog box native to your platform, and thus can differ from what you see in the examples that follow.

For UNIX platforms, the dialog box is like the one shown in the following figure.



For Mac platforms, the dialog box is like the one shown in the following figure.

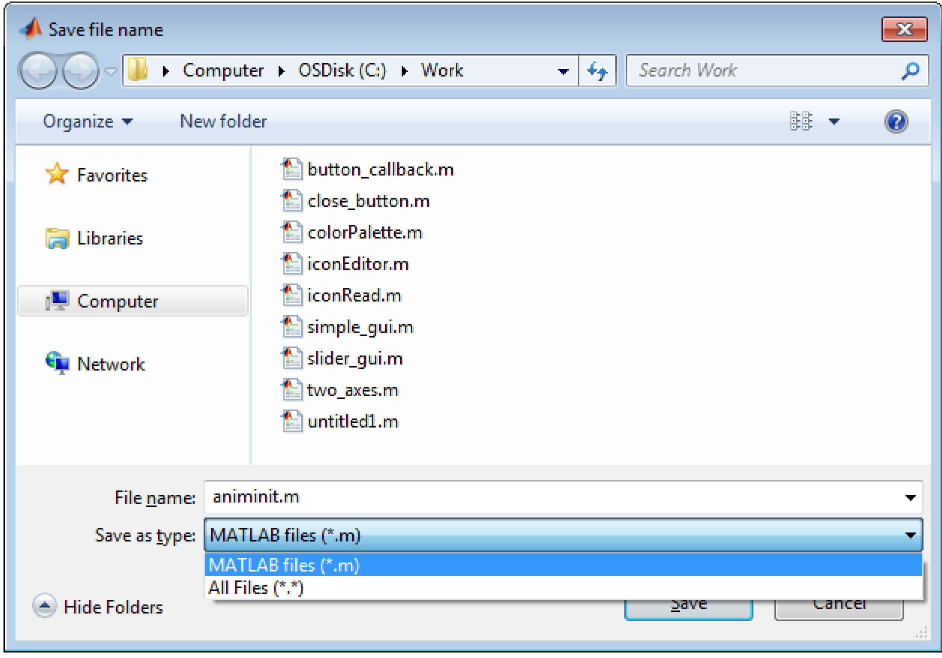


Note A modal dialog box prevents you from interacting with other MATLAB windows before responding. To block MATLAB program execution as well, use the `uiwait` function. For more information about modal dialog boxes, see `WindowStyle` in MATLAB Figure Properties.

Examples

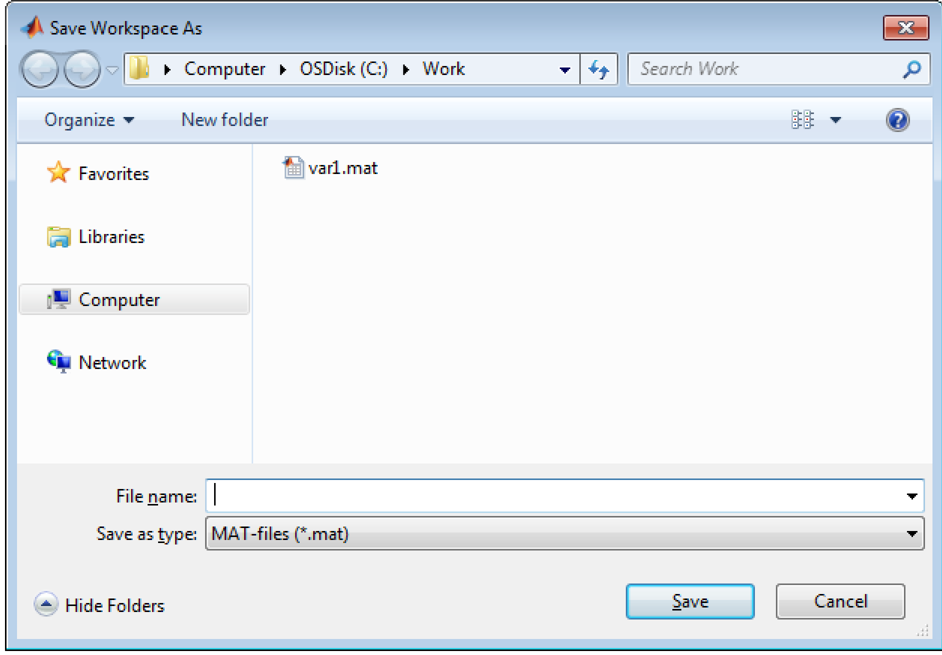
The following statement displays a dialog box entitled `'Save file name'`, setting the **File name** field to `animinit.m` and the filter to `program files (*.m)`. Because `FilterSpec` is a string, the filter also includes `All Files (*.*)`

```
[file,path] = uiputfile('animinit.m','Save file name');
```



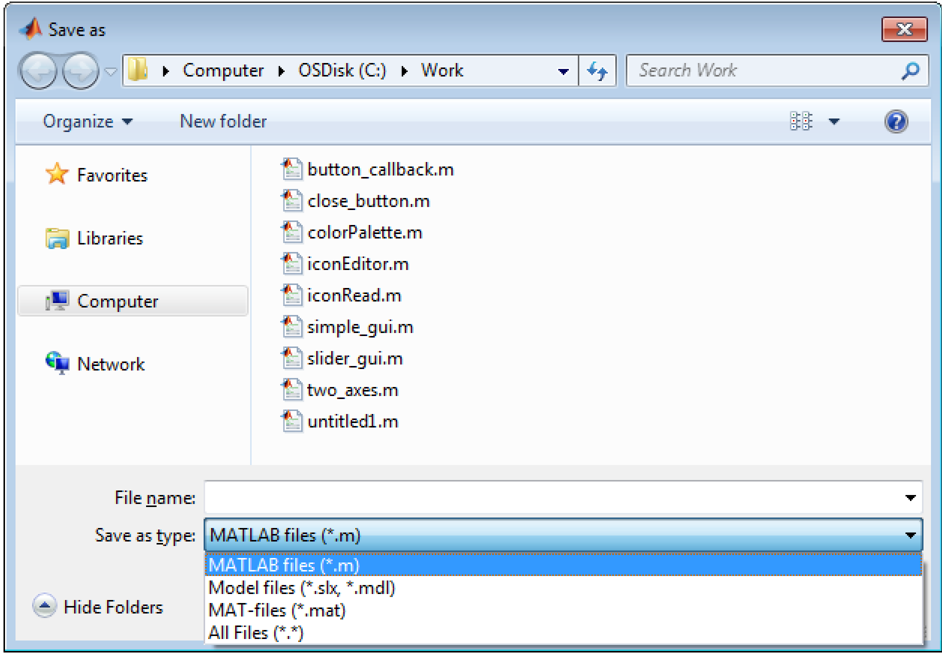
The following statement displays a dialog box entitled `'Save Workspace As'` with the filter set to MAT-files.

```
[file,path] = uiputfile('*.mat','Save Workspace As');
```



To display several file types in the **Save as type** list box, separate each file extension with a semicolon, as in the following code. `uiputfile` displays a default description for each known file type, such as "Model files" for Simulink `.mdl` and `.slx` files.

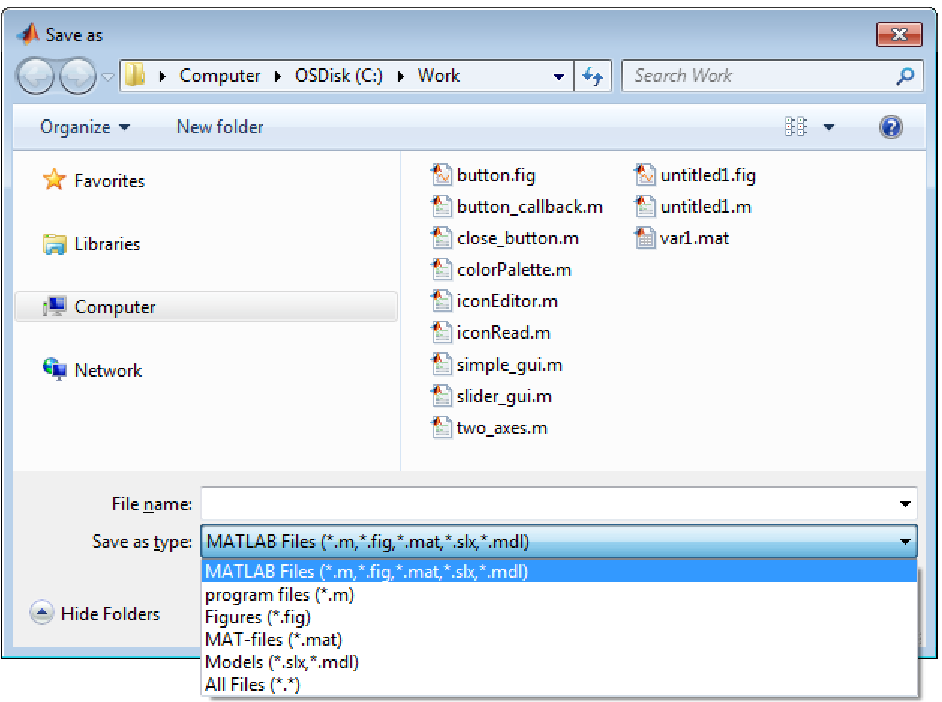
```
[filename, pathname] = uiputfile(...
 {'*.m';'*.slx';'*.mat';'*.*'},...
 'Save as');
```



To create a list of file types and give them descriptions that are different from the defaults, use a cell array. This example also associates multiple file types with the `'MATLAB Files'` and `'Models'` descriptions.

```
[filename, pathname, filterindex] = uiputfile( ...
{'*.m;*.fig;*.mat;*.slx;*.mdl',...
 'MATLAB Files (*.m,*.fig,*.mat,*.slx,*.mdl)';
 '*.m', 'program files (*.m)';...
 '*.fig','Figures (*.fig)';...
 '*.mat','MAT-files (*.mat)';...
 '*.slx;*.mdl','Models (*.slx,*.mdl)';...
 '*.*',  'All Files (*.*)'},...
 'Save as');
```

The first column of the cell array contains the file extensions, while the second contains the descriptions you want to provide for the file types. The first entry of column one contains several extensions separated by semicolons. These file types all associate with the description `'MATLAB Files (*.m,*.fig,*.mat,*.slx,*.mdl)'`. The code produces the dialog box shown in the following figure.



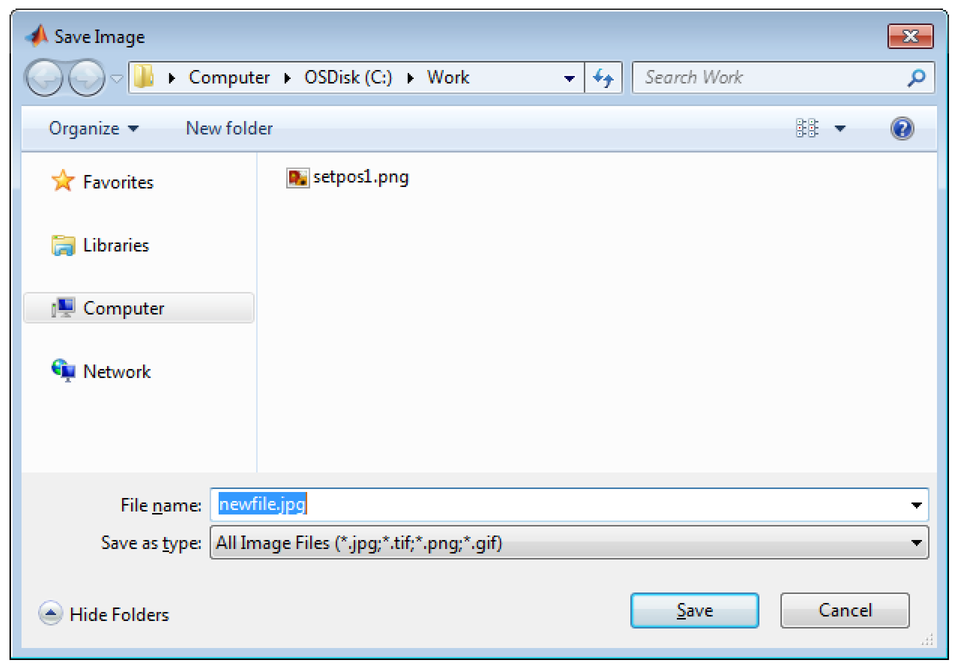
The following code checks for the existence of the file and displays a message about the result of the file selection operation.

```
[filename, pathname] = uiputfile('*.m','Pick a MATLAB program file');
if isequal(filename,0) || isequal(pathname,0)
   disp('User selected Cancel')
else
```

```
   disp(['User selected ',fullfile(pathname,filename)])
end
```

Select or enter a file name for saving a figure as an image in one of four formats, described in a cell array.

```
uiputfile({'*.jpg;*.tif;*.png;*.gif','All Image Files';...
          '*.*','All Files' },'Save Image',...
          'C:\Work\newfile.jpg')
```



See Also

save | uigetdir | uigetfile | uisave

Purpose

Resume execution of blocked program

Syntax

```
uiresume(h)
```

Description

`uiresume(h)` resumes the program execution that `uiwait` suspended.

Tips

The `uiwait` and `uiresume` functions block and resume MATLAB program execution. When creating a dialog box, you should have a uicontrol component with a callback that calls `uiresume` or a callback that destroys the dialog box. These are the only methods that resume program execution after the `uiwait` function blocks execution.

When used in conjunction with a modal dialog box, `uiresume` can resume the execution of the program that `uiwait` suspended while presenting a dialog box.

Examples

This example creates a GUI with a **Continue** push button. The example calls `uiwait` to block MATLAB execution until `uiresume` is called. This happens when the user clicks the **Continue** push button because the push button's `Callback`, which responds to the click, calls `uiresume`.

```
f = figure;
h = uicontrol('Position',[20 20 200 40],'String','Continue',...
              'Callback','uiresume(gcbf)');
disp('This will print immediately');
uiwait(gcf);
disp('This will print after you click Continue');
close(f);
```

`gcbf` is the handle of the figure that contains the object whose callback is executing.

"Modal Dialog Box (GUIDE)" is a more complex example for a GUIDE GUI.

See Also

`dialog` | `figure` | `uicontrol` | `uimenu` | `uiwait` | `waitfor`

uisave

Purpose

Interactively save workspace variables to MAT-file

Syntax

```
uisave
uisave(variables)
uisave(variables,filename)
```

Description

`uisave`, with no arguments, prompts you for a file name, and then saves all variables from your workspace to that file.

`uisave(variables)` saves one or more workspace variables listed in `variables`.

`uisave(variables,filename)` uses the specified `filename` as the default **File name** in the Save Workspace Variables dialog box, instead of the default `matlab.mat`.

Tips

- If you type a name in the **File name** field, such as `my_vars`, and click **Save**, the dialog saves all workspace variables to the file `my_vars.mat`. The default filename is `matlab.mat`.
- `uisave` calls `uiputfile` to choose a filename.
- If the filename you specify exists in that folder, `uisave` informs you and gives you a chance to cancel the operation.
- The `uisave` dialog box is modal.
- The `uisave` dialog is modal. A modal dialog box prevents you from interacting with other windows until you respond to it.

Input Arguments

variable

String containing the name of a variable in the current workspace or cell array of strings when specifying more than one variable.

> **Default:** All variables in the current workspace are saved to a MAT-file.

filename

String naming a file to appear in the dialog **File name** field when the dialog opens. You can omit a file extension, or specify the file extension as `.mat`.

Default: `matlab.mat`

Definitions

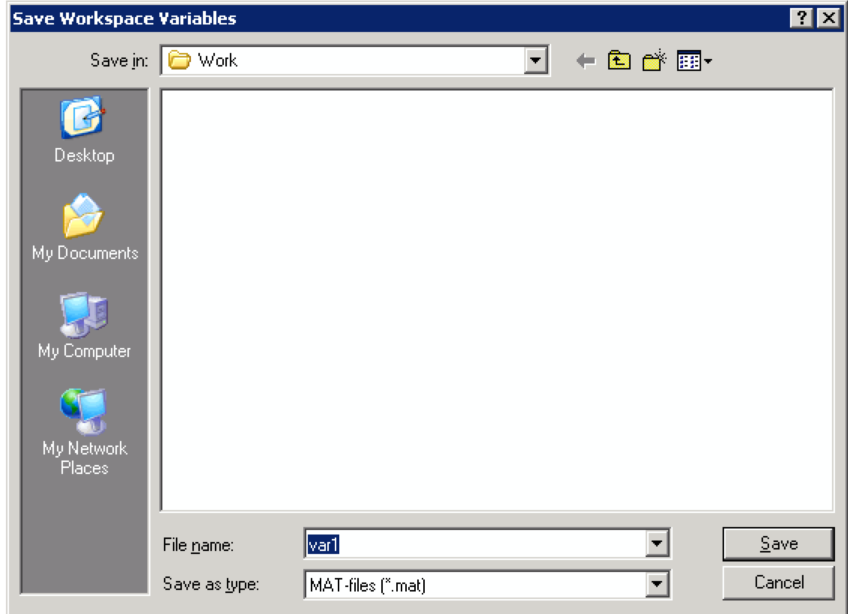
Modal Dialog

A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

Examples

Create workspace variables `h` and `g`, and then display the Save Workspace Variables dialog box in the current folder with the default **File name** set to `var1`.

```
h = 365;
g = 52;
uisave({'h','g'},'var1');
```



Clicking **Save** stores the workspace variables `h` and `g` in the file `var1.mat` in the displayed folder.

Alternatives

Use any of the following GUI options to save workspace variables:

- Use **File > Save** to save workspace variables.
- Click the Save icon in the Workspace Browser.
- Select one or more variables in the Workspace Browser, right-click, and choose **Save as** from the context menu.

See Also

`save` | `uigetfile` | `uiputfile` | `uiopen`

Tutorials

- "Save, Load, and Delete Workspace Variables"

Purpose

Open standard dialog box for setting object's ColorSpec

Syntax

```
c = uisetcolor
c = uisetcolor([r g b])
c = uisetcolor(h)
c = uisetcolor(...,'dialogTitle')
```

Description

`c = uisetcolor` displays a modal color selection dialog appropriate to the platform, and returns the color selected by the user. The dialog box is initialized to `white`.

`c = uisetcolor([r g b])` displays a dialog box initialized to the specified color, and returns the color selected by the user. `r`, `g`, and `b` must be values between 0 and 1.

`c = uisetcolor(h)` displays a dialog box initialized to the color of the object specified by handle `h`, returns the color selected by the user, and applies it to the object. `h` must be the handle to an object containing a color property.

`c = uisetcolor(...,'dialogTitle')` displays a dialog box with the specified title.

If the user presses **Cancel** from the dialog box, or if any error occurs, the output value is set to the input RGB triple, if provided; otherwise, it is set to 0.

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

See Also

`ColorSpec`

uisetfont

Purpose

Open standard dialog box for setting object's font characteristics

Syntax

```
uisetfont
uisetfont(h)
uisetfont(S)
uisetfont(...,'DialogTitle')
S = uisetfont(...)
```

Description

`uisetfont` enables you to change font properties (`FontName`, `FontUnits`, `FontSize`, `FontWeight`, and `FontAngle`) for a `text`, `axes`, or `uicontrol` object. The function returns a structure consisting of font properties and values. You can specify an alternate title for the dialog box.

`uisetfont` displays a modal dialog box and returns the selected font properties.

Note A modal dialog box prevents the user from interacting with other windows before responding. For more information, see `WindowStyle` in the MATLAB Figure Properties.

`uisetfont(h)` displays a modal dialog box, initializing the font property values with the values of those properties for the object whose handle is `h`. Selected font property values are applied to the current object. If a second argument is supplied, it specifies a name for the dialog box.

`uisetfont(S)` displays a modal dialog box, initializing the font property values with the values defined for the specified structure (`S`). `S` must define legal values for one or more of these properties: `FontName`, `FontUnits`, `FontSize`, `FontWeight`, and `FontAngle` and the field names must match the property names exactly. If other properties are defined, they are ignored. If a second argument is supplied, it specifies a name for the dialog box.

`uisetfont(...,'DialogTitle')` displays a modal dialog box with the title `DialogTitle` and returns the values of the font properties selected in the dialog box.

`S = uisetfont(...)` returns the properties `FontName`, `FontUnits`, `FontSize`, `FontWeight`, and `FontAngle` as fields in a structure. If the user presses **Cancel** from the dialog box or if an error occurs, the output value is set to 0.

Examples

These statements create a text object, then display a dialog box (labeled Update Font) that enables you to change the font characteristics:

```
h = text(.5,.5,'Figure Annotation');
uisetfont(h,'Update Font')
```

These statements create two push buttons, then set the font properties of one based on the values set for the other:

```
% Create push button with string ABC
c1 = uicontrol('Style', 'pushbutton', ...
     'Position', [10 10 100 20], 'String', 'ABC');
% Create push button with string XYZ
c2 = uicontrol('Style', 'pushbutton', ...
     'Position', [10 50 100 20], 'String', 'XYZ');
% Display set font dialog box for c1, make selections,
& and save to d
d = uisetfont(c1);
% Apply those settings to c2
set(c2, d)
```

See Also

`axes` | `text` | `uicontrol`

uisetpref

Purpose Manage preferences used in `uigetpref`

Syntax

```
uisetpref('clearall')
```

Description `uisetpref('clearall')` resets the value of all preferences registered through `uigetpref` to `'ask'`. This causes the dialog box to display when you call `uigetpref`.

Note Use `setpref` to set the value of a particular preference to `'ask'`.

See Also `setpref` | `uigetpref`

Purpose

Reorder visual stacking order of objects

Syntax

```
uistack(h)
uistack(h,stackopt)
uistack(h,stackopt,step)
```

Description

`uistack(h)` raises the visual stacking order of the objects specified by the handles in `h` by one level (`step` of 1). All handles in `h` must have the same parent.

`uistack(h,*stackopt*)` moves the objects specified by `h` in the stacking order, where *`stackopt`* is one of the following:

- `'up'` – moves `h` up one position in the stacking order
- `'down'` – moves `h` down one position in the stacking order
- `'top'` – moves `h` to the top of the current stack
- `'bottom'` – moves `h` to the bottom of the current stack

`uistack(h,*stackopt*,step)` moves the objects specified by `h` up or down the number of levels specified by `step`.

Note In a GUI, `axes` objects are always at a lower level than `uicontrol` objects. You cannot stack an `axes` object on top of a `uicontrol` object.

See "Set Tab Order in a Programmatic GUI" in the MATLAB documentation for information about changing the tab order.

Examples

The following code moves the child that is third in the stacking order of the figure handle `hObject` down two positions.

```
v = allchild(hObject)
uistack(v(3),'down',2)
```

uitable

Purpose

Create 2-D graphic table GUI component

Syntax

```
uitable
uitable('PropertyName1', value1,'PropertyName2',value2,...)
uitable(parent,...)
handle = uitable(...)
```

Description

`uitable` creates an empty `uitable` object in the current figure window, using default property values. If no figure exists, a new figure window opens.

`uitable('PropertyName1', value1,'PropertyName2',value2,...)` creates a `uitable` object with specified property values. Properties that you do not specify assume the default property values. See the Uitable Properties reference page for information about the available properties.

`uitable(parent,...)` creates a `uitable` object as a child of the specified parent handle `parent`. The parent can be a figure or `uipanel` handle. If you also specify a different value for the `Parent` property, the value of the `Parent` property takes precedence.

`handle = uitable(...)` creates a `uitable` object and returns its handle.

Tips

After creating a `uitable` object, you can set and query its property values using the `set` and `get` functions.

If the `ColumnEditable` property is `true` for columns you edit, you can change values in a displayed table. By default, this property is `false` for all columns. If a noneditable column contains pop-up choices, only the current choice is visible (and not the pop-up menu control).

If you attempt to create a `uitable` object when running MATLAB on a UNIX[9] system without a Java virtual machine (`matlab -nojvm`) or without a display (`matlab  nodisplay`), no table generates and you receive an error.

9. UNIX is a registered trademark of The Open Group in the United States and other countries.

The **CellEditCallback** executes after you edit a value and do any of the following:

- Type **Enter**.
- Click another table cell.
- Click anywhere else within the table.
- Click another control or area within the same figure window.
- Click another window, click again on the GUI containing the table (or use **Alt+Tab** to switch windows), and then perform any of the above four actions.

When the `CellEditCallback` callback executes, `uitable` updates the underlying data matrix (the table `Data` property) to contain the value that the cell now displays.

The **CellSelectionCallback** executes when you select a table cell or remove one from the current selection by **Ctrl**+clicking it. Clicking a cell without pressing any key selects it and deselects all currently selected cells. You can define a range of table cells by **Shift**+clicking an unselected cell after selecting one or more cells. The callback provides event data that identifies the rows and columns of all cells in the current selection.

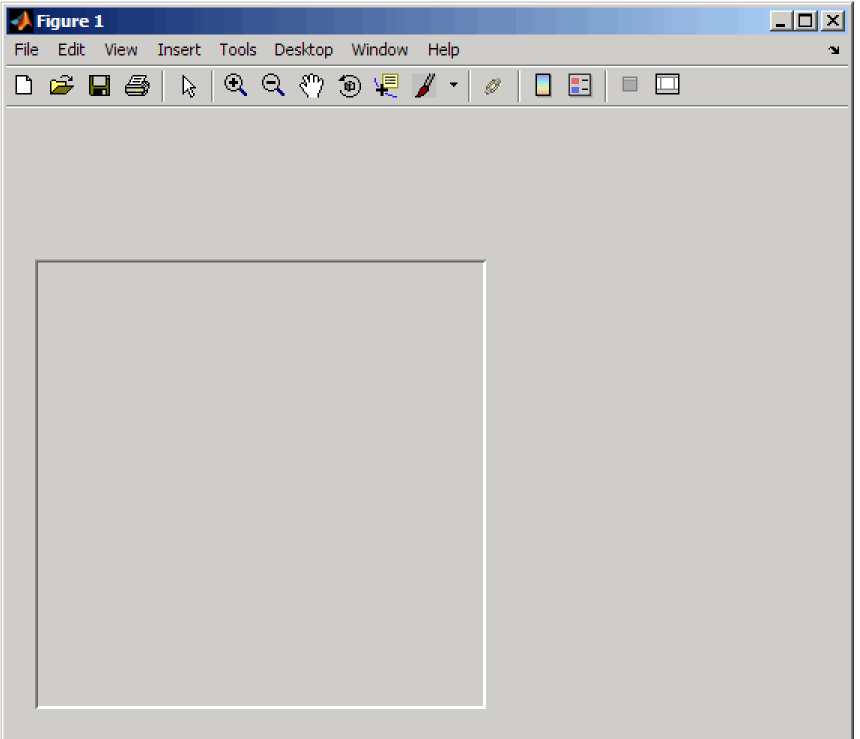
You cannot select table cells programmatically. Directly clicking cells is the only method of selection.

Examples

Create a table, provide magic-square data, set column widths uniformly, and specify the uitable `ColumnWidth` property as a cell array:

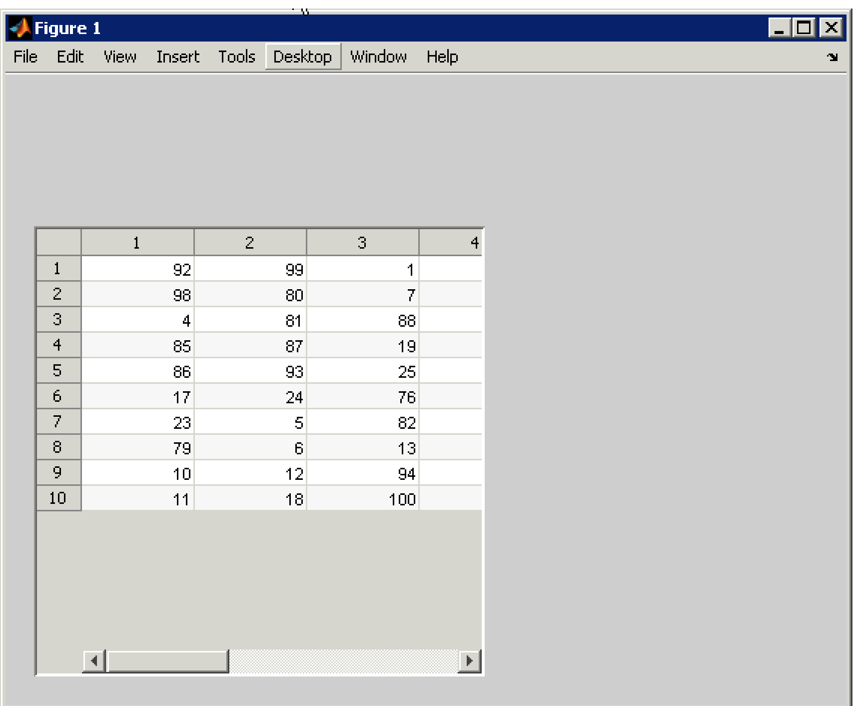
1. Create a table in the current figure. If no figure exists, one opens:

```
t = uitable;
```



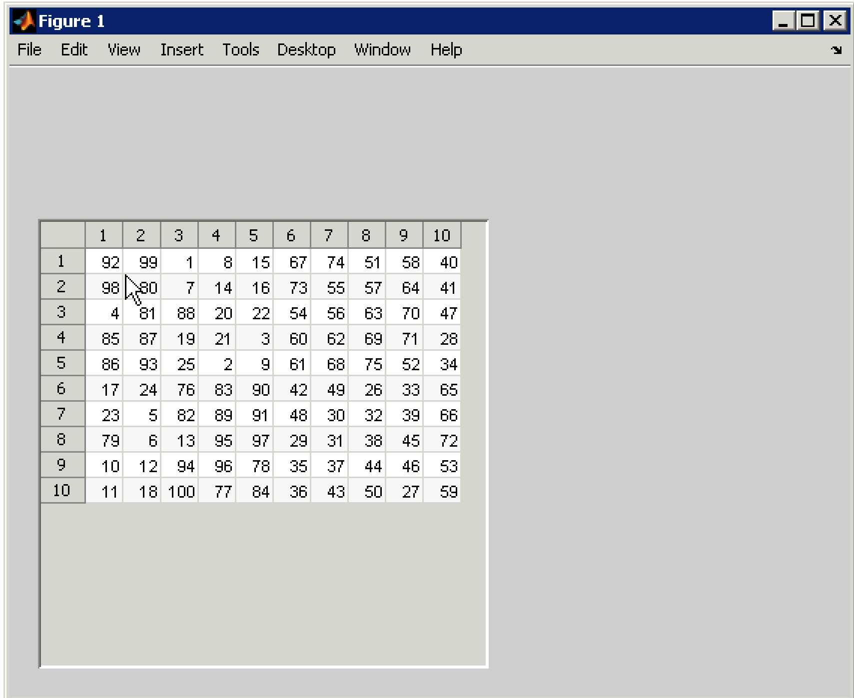
2 As the table has no content (its `Data` property is empty), it initially displays no rows or columns. Provide data (a magic square)

```
set(t,'Data',magic(10))
```



3 Make the entire table contents visible. Set column widths to 25 pixels uniformly. Specify the `ColumnWidth` property of the table as a cell array.

```
set(t,'ColumnWidth',{25})
```



	1	2	3	4	5	6	7	8	9	10
1	92	99	1	8	15	67	74	51	58	40
2	98	80	7	14	16	73	55	57	64	41
3	4	81	88	20	22	54	56	63	70	47
4	85	87	19	21	3	60	62	69	71	28
5	86	93	25	2	9	61	68	75	52	34
6	17	24	76	83	90	42	49	26	33	65
7	23	5	82	89	91	48	30	32	39	66
8	79	6	13	95	97	29	31	38	45	72
9	10	12	94	96	78	35	37	44	46	53
10	11	18	100	77	84	36	43	50	27	59

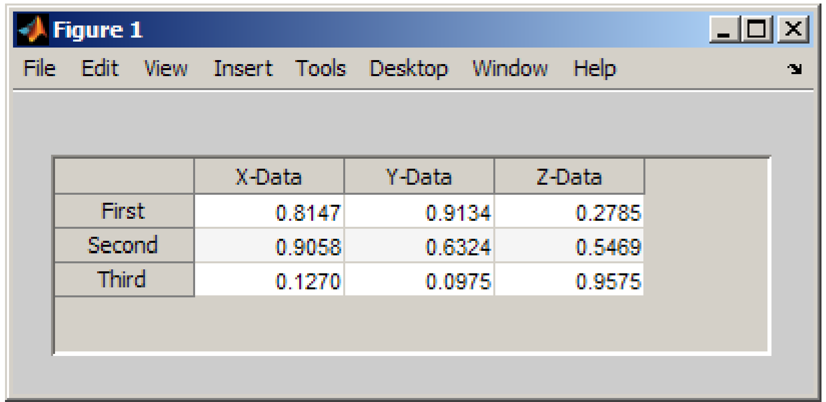
Cell arrays that specify `ColumnWidth` can contain:

- One number (a width measured in pixels, as shown here) or the string `'auto'`.
- A cell array containing a list of pixel sizes having up to as many entries as the table has columns .

If a list of column widths has `n` entries, where `n` is smaller than the number of columns, it sets the first `n` column widths only. You can substitute `'auto'` for any value in the cell array to have the width of that column calculated automatically.

Create a figure and add a table to contain a 3-by-3 data matrix. The code specifies the column names, row names, parent, and position of the table:

```
f = figure('Position',[200 200 400 150]);
dat = rand(3);
cnames = {'X-Data','Y-Data','Z-Data'};
rnames = {'First','Second','Third'};
t = uitable('Parent',f,'Data',dat,'ColumnName',cnames,...
            'RowName',rnames,'Position',[20 20 360 100]);
```

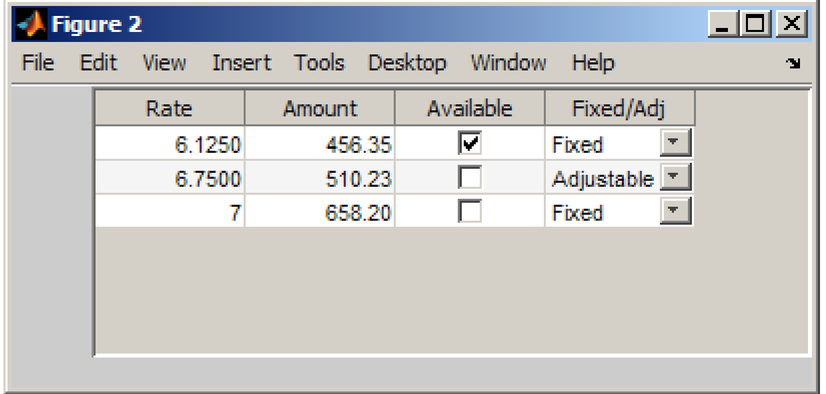


Create a table to contain a 3-by-4 array that contains numeric, logical, and string data, as follows:

- First column (**Rate**): Numeric, with three decimals (not editable)
- Second column (**Amount**): Currency (not editable)
- Third column (**Available**): Check box (editable)
- Fourth column (**Fixed/Adj**): Pop-up menu with two choices: `Fixed` and `Adjustable` (editable)
- Specify the `RowName` property as empty to remove row names from the table.

```
f = figure('Position',[100 100 400 150]);
```

```
dat =  {6.125, 456.3457, true,  'Fixed';...
        6.75,  510.2342, false, 'Adjustable';...
        7,     658.2,    false, 'Fixed';};
columnname =   {'Rate', 'Amount', 'Available', 'Fixed/Adj'};
columnformat = {'numeric', 'bank', 'logical', {'Fixed' 'Adjustable'}};
columneditable =  [false false true true];
t = uitable('Units','normalized','Position',...
            [0.1 0.1 0.9 0.9], 'Data', dat,...
            'ColumnName', columnname,...
            'ColumnFormat', columnformat,...
            'ColumnEditable', columneditable,...
            'RowName',[]);
```



Alternatives

You can add tables to GUIs you create with "Table".

See Also

`figure` | `format` | `get` | `set` | `uipanel` | Uitable Properties

Tutorials

- "GUI for Presenting Data in Multiple, Synchronized Displays (GUIDE)"
- "GUI for Presenting Data in Multiple, Synchronized Displays"

How To

- “Table”

Uitable Properties

Purpose

Describe table properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

You can set default uitable properties by typing:

```
set(h,'DefaultUitablePropertyName',PropertyValue...)
```

Where `h` can be the root handle (`0`), a figure handle, or a uitable handle. *`PropertyName`* is the name of the uitable property and `PropertyValue` is the value you specify as the default for that property.

For more information about changing the default value of a property see “Setting Default Property Values”. For an example, see the `CreateFcn` property.

Uitable Properties

This section lists all properties useful to `uitable` objects along with valid values and descriptions of their use. In the property descriptions, curly braces { } enclose default values.

Property Name	Description
`BackgroundColor`	Background color of cells.
`BeingDeleted`	This object is being deleted.
`BusyAction`	Callback routine interruption
`ButtonDownFcn`	Button-press callback routine
`CellEditCallback`	Callback when data in a cell is changed.

Property Name	Description
CellSelectionCallback	Callback when cell is selected
Children	uitable objects have no children
Clipping	Does not apply to uitable objects
ColumnEditable	Determines data in a column as editable
ColumnFormat	Determines display and editablility of columns
ColumnName	Column header label
ColumnWidth	Width of each column in pixels
CreateFcn	Callback routine during object creation
Data	Table data
DeleteFcn	Callback routine during object deletion
Enable	Enable or disable the uitable
Extent	Size of uitable rectangle
FontAngle	Character slant of cell content
FontName	Font family for cell content
FontSize	Font size of cell content
FontUnits	Font size units for cell content
FontWeight	Weight of cell text characters
ForegroundColor	Color of text in cells
HandleVisibility	Control access to object's handle
HitTest	Selectable by mouse click
Interruptible	Callback routine interruption mode
KeyPressFcn	Key press callback function
Parent	uitable parent

Property Name	Description
Position	Size and location of uitable
RearrangeableColumns	Location of the column
RowName	Row header label names
RowStriping	Color striping of label rows
Selected	Is object selected?
SelectionHighlight	Object highlight when selected
Tag	Use-specified object label
TooltipString	Content of tooltip for object
Type	Class of graphics object
UIContextMenu	Associate context menu with uitable
Units	Units of measurement
UserData	User-specified data
Visible	uitable visibility

BackgroundColor
    1-by-3 or 2-by-3 matrix of RGB triples

Cell background color. Color used to fill the uitable cells. Specify as an 1-by-3 or 2-by-3 matrix of RGB triples, such as [.8 .9. .8] or [1 1 .9; .9 1 1]. Each row is an RGB triplet of real numbers between 0.0 and 1.0 that defines one color. (Color names are not allowed.) The default is a 1-by-3 matrix of platform-dependent colors. See ColorSpec for information about RGB colors.

Row 2 of the matrix is used only if the `RowStriping` property is `on`. The table background is not striped unless both `RowStriping` is on and the `BackgroundColor` color matrix has two rows.

`BeingDeleted`

`on | {off}` (read-only)

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB software sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property) It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, some functions may not need to perform actions on objects that are being deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`ButtonDownFcn`

string or function handle (GUIDE sets this property)

Button-press callback routine. A callback routine that can execute when you press a mouse button while the pointer is on or near a `uitable`. Specifically:

- If the `uitable` `Enable` property is set to `on`, the `ButtonDownFcn` callback executes when you click the right or left mouse button in a 5-pixel border around the `uitable` or when you click the right mouse button on the control itself.
- If the `uitable` `Enable` property is set to `inactive` or `off`, the `ButtonDownFcn` executes when you click the right or left mouse button in the 5-pixel border or on the control itself.

This is useful for implementing actions to interactively modify control object properties, such as size and position, when they are clicked on.

Define this routine as a string that is a valid MATLAB expression or the name of a MATLAB function file. The expression executes in the MATLAB workspace.

To add a `ButtonDownFcn` callback in GUIDE, select **View Callbacks** from the Layout Editor **View** menu, then select `ButtonDownFcn`. GUIDE sets this property to the appropriate string and adds the callback to the program file the next time you save the GUI. Alternatively, you can set this property to the string `%automatic`. The next time you save the GUI, GUIDE sets this property to the appropriate string and adds the callback to the program file.

`CellEditCallback`

function handle, cell array containing function handle and additional arguments, or string (not recommended)

Callback to edit user-entered data

Callback function executed when the user modifies a table cell. It can perform evaluations, validations, or other customizations. If this function is called as a function handle, `uitable` passes it two arguments. The first argument, `source`, is the handle of the `uitable`. The second argument, `eventdata`, is an event data structure that contains the fields shown in the following table. All fields in the event data structure are read only.

Event Data Structure Field	Type	Description
`Indices`	1-by-2 matrix	Row index and column index of the cell the user edited.
`PreviousData`	1-by-1 matrix or cell array	Previous data for the changed cell. The default is an empty matrix, `[]`.
`EditData`	String	User-entered string.

Event Data Structure Field	Type	Description
NewData	1-by-1 matrix or cell array	Value that uitable wrote to Data. It is either the same as EditData or a converted value, for example, 2 where EditData is '2' and the cell is numeric. Empty if uitable detected an error in the user-entered data and did not write it to Data.
Error	String	Error that occurred when uitable tried to convert the EditData string into a value appropriate for Data. For example, uitable could not convert the EditData string consistent with the Column Format property, if any, or the data type for the changed cell. Empty if uitable wrote the value to Data. If Error is not empty, the CellEditCallback can pass the error string to the user or can attempt to manipulate the data. For example, the string 'pi' would raise an error in a numeric cell but the CellEditCallback could convert it to its numerical equivalent and store it in Data without passing the error to the user.

When a user edits a cell, uitable first attempts to store the user-entered value in Data, converting the value if necessary. It then calls the CellEditCallback and passes it the event data structure. If there is no CellEditCallback and the user-entered data results it an error, the contents of the cell reverts to its previous value and no error is displayed. The CellEditCallback

is issued when the user has modified a table cell and presses **Enter** or clicks anywhere outside the cell.

`CellSelectionCallback`

function handle, cell array containing function handle and additional arguments, or string (not recommended)

Callback that executes when cell is selected. Callback function that executes when the user highlights a cell by navigating to it or clicking it. For multiple selection, this callback executes when new cells are added to the selection. The callback includes event data, a structure with one member

Event Data Structure Field	Type	Description
`Indices`	n-by-2 matrix	Row index and column index of the cells the user currently has selected

Once a cell selection has been made, cells within it can be removed one at a time by **Ctrl**-clicking them.

`Children`

matrix

The empty matrix; `uitable` objects have no children.

`Clipping`

`{on} | off`

This property has no effect on `uitable` objects.

`ColumnEditable`

logical 1–by-n matrix | scalar logical value |{ empty matrix ([])}

Determines if column is user-editable.

Determines if the data can be edited by the end user. Each value in the cell array corresponds to a column. False is default because the developer needs to have control over changes users potentially might make to data.

Specify elements of a logical matrix as `true` if the data in a column is editable by the user or `false` if it is not. An empty matrix indicates that no columns are editable.

Columns that contain check boxes or pop-up menus must be editable for the user to manipulate these controls. If a column that contains pop-up menus is not editable, the currently selected choice appears without displaying the pop-up control. The Elements of the `ColumnEditable` matrix must be in the same order as columns in the `Data` property. If you do not specify `ColumnEditable`, the default is an empty matrix (`[]`).

`ColumnFormat`
cell array of strings

Cell display formatting. Determines how the data in each column displays and is edited. Elements of the cell array must be in the same order as table columns in the `Data` property. If you do not want to specify a display format for a particular column, enter `[]` as a placeholder. If no format is specified for a column, the default display is determined by the data type of the data in the cell. Default `ColumnFormat` is an empty cell array (`{}`). In most cases, the default is similar to the command window.

Elements of the cell array must be one of the strings described in the following table.

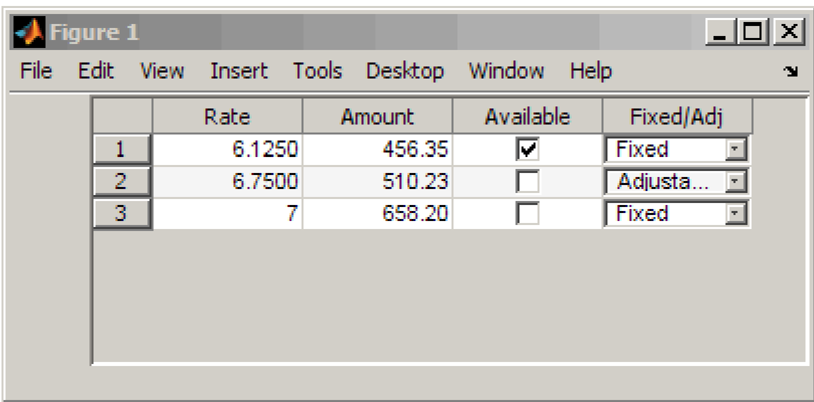
Cell Format	Description
`'char'`	Displays a left-aligned string. To edit, the user types a string that replaces the existing string.
`'logical'`	Displays a check box. To edit, the user checks or unchecks the check box. `uitable` sets the corresponding `Data` value to `true` or `false` accordingly. Initially, the check box is checked if the corresponding `Data` value would produce
`'numeric'`	Displays a right-aligned string equivalent to the command window, for numeric data. If the cell `Data` value is boolean, then 1 or 0 is displayed. If the cell `Data` value is not numeric and not boolean, then `NaN` is displayed. To edit, the user can enter any string. This enables a user to enter a value such as `'pi'` that can be converted to its numeric equivalent by a `CellEditCallback`. The `uitable` function first attempts to convert the user-entered string to a numeric value and store it in `Data`. It then calls the `CellEditCallback`. See `CellEditCallback` for more information.

Cell Format	Description
1–by-n cell array of strings that define a pop-up menu, e.g., {'one' 'two' 'three'}	Displays a pop-up menu. To edit, the user makes a selection from the pop-up menu. uitable sets the corresponding Data value to the selected menu item. The initial values for the pop-up menus in the column are the corresponding strings in Data. These initial values do not have to be items in the pop-up menu. See Example 3 on the uitable reference page.
Valid string accepted by the format function, e.g.,'short' or 'bank'	Displays the Data value using the specified format. For example, for a two-column table, set(htable,'ColumnFormat',{'short','bank'}).

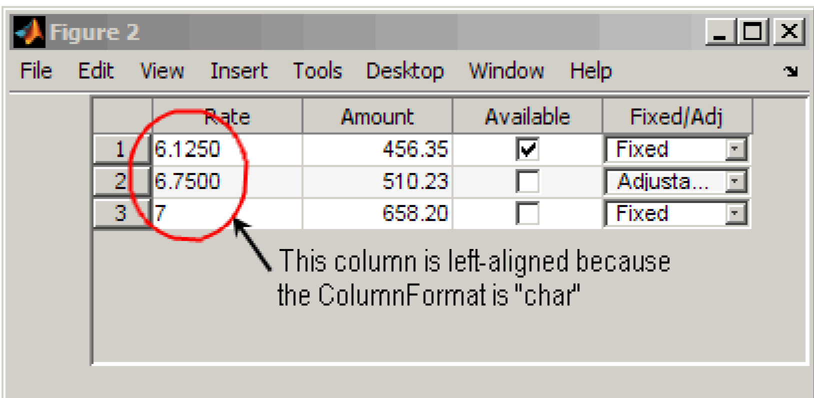
In some cases, you may need to insert an appropriate column in Data. If Data is a numerical or logical matrix, you must first convert it to a cell array using the mat2cell function.

Data and ColumnFormat

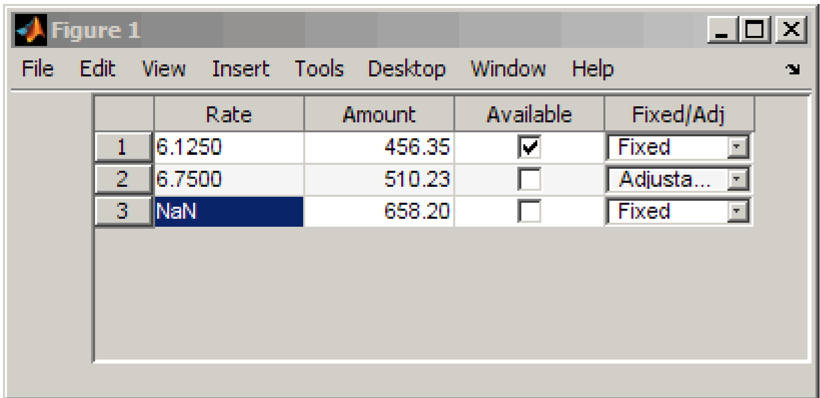
When you create a table, you must specify value of Data. The Data property dictates what type of data can exist in any given cell. By default, the value of the Data also dictates the display of the cell to the end user, unless you specify a different format using the ColumnFormat property.



`ColumnFormat` controls the presentation of the `Data` to the end user. Therefore, if you specify a `ColumnFormat` of `char` (or pick **Text** from the Table Property Editor), you are asking the table to display the `Data` associated with that column as a string. For example, if the `Data` for a particular column is `numeric`, and you specify the `ColumnFormat` as `char`, then the display of the numeric data will be left-aligned



If your column is editable and the user enters a number, the number will be left-aligned. However, if the user enters a text string, the table displays a **NaN**.



Another possible scenario is that the value `Data` is `char` and you set the `ColumnFormat` to be a pop-up menu. Here, if the value of the `Data` in the cell matches one of the pop-up menu choices you define in `ColumnFormat`, then the `Data` is shown in the cell. If it does not match, then the cell defaults to display the first option from the choices you specify in `ColumnFormat`. Similarly, if `Data` is numeric or logical with the `ColumnFormat` as pop-up menu, if the `Data` value in the cell does not match any of the choices you specify in `ColumnFormat`, the cell defaults to display the first option in the pop-menu choice.

This table describes how `Data` values correspond with your `ColumnFormat` when the columns are editable.

	`ColumnFormat` Selections		
	`numeric`	`char`	`logical`

Data Type	`numeric`	Values match. MATLAB displays numbers as is.	MATLAB converts the text string entered to a double. See `str2double` for more information. If string cannot be converted, **NaN** is displayed.	Does not work: warning is thrown. **Note** If you have defined `CellEditCallback`, this warning will not be thrown
	`char`	MATLAB converts the entered number to a text string.	Values match. MATLAB displays the string as is.	Does not work: warning is thrown. **Note** If you have defined `CellEditCallback`, this warning will not be thrown
	`logical`	Does not work: warning is thrown. **Note** If you have defined `CellEditCallback`, this warning will not be thrown	If text string entered is `true` or `false`, MATLAB converts string to the corresponding logical value and displays it. For all others, it Does not work: warning is thrown.	Values match. MATLAB displays logical value as a check box as is.

			Note If you have defined `CellEditCallback`, this warning will not be thrown	

If you get a mismatch error, you have the following options:

- Change the `ColumnFormat` or value of `Data` to match.
- Implement the `CellEditCallback` to handle custom data conversion.

`ColumnName`

1–by-*n* cell array of strings`|` {`'numbered'`} `|` empty matrix (`[]`)

Column heading names. Each element of the cell array is the name of a column. Multiline column names can be expressed as a string vector separated by vertical slash (`|`) characters, e.g.,`'Standard|Deviation'`

For sequentially numbered column headings starting with `1`, specify `ColumnName` as `'numbered'`. This is the default.

To remove the column headings, specify `ColumnName` as the empty matrix (`[]`).

The number of columns in the table is the larger of `ColumnName` and the number of columns in the `Data` property matrix or cell array.

`ColumnWidth`

1–by-*n* cell array or `'auto'`

Column widths. The width of each column in units of pixels. Column widths are always specified in pixels; they do not obey the `Units` property. Each column in the cell array corresponds to a column in the uitable. By default, the width of the column name, as specified in `ColumnName`, along with some other factors, is used to determine the width of a column. If `ColumnWidth` is a cell array and the width of a column is set to `'auto'`or if **auto** is selected for that column in the Property Inspector GUI for columns, the column width defaults to a size determined by the table. The table decides the default size using a number of factors, including the `ColumnName` and the minimum column size.

To default all column widths in an existing table, use

```
set(uitable_handle,'ColumnWidth','auto')
```

To default some column widths but not others, use a cell array containing a mixture of pixel values and `'auto'`. For example,

```
set(uitable_handle,'ColumnWidth',{64 'auto' 40 40 'auto' 72})
```

`CreateFcn`

string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a `uitable` object. MATLAB sets all property values for the uitable before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the uitable being created.

Setting this property on an existing uitable object has no effect.

You can define a default `CreateFcn` callback for all new uitables. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uitable`. For example, the code

```
set(0,'DefaultUitableCreateFcn','set(gcbo,...
```

```
  ''BackGroundColor'',''blue'')')
```

creates a default `CreateFcn` callback that runs whenever you create a new uitable. It sets the default background color of all new uitables.

To override this default and create a uitable whose `BackgroundColor` is set to a different value, call `uitable` with code similar to

```
hpt = uitable(...,'CreateFcn','set(gcbo,...
''BackgroundColor'',''white'')')
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uitable` call. In the example above, if instead of redefining the `CreateFcn` property for this uitable, you had explicitly set `BackgroundColor` to `white`, the default `CreateFcn` callback would have set `BackgroundColor` back to the default, i.e., `blue`.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

See "Function Handle Callbacks" for information on how to use function handles to define a callback function.

`Data`

matrix or cell array of numeric, logical, or character data

Data content of uitable. The matrix or cell array must be 2–dimensional. A cell array can mix data types.

Use `get` and `set` to modify `Data`. For example,

```
data = get(tablehandle,'Data')
data(event.indices(1),event.indices(2)) = pi();
set(tablehandle,'Data',data);
```

See `CellEditCallback` for information about the event data structure. See `ColumnFormat` for information about specifying the data display format.

The number of rows in the table is the larger of `RowName` and the number of rows in `Data`. The number of columns in the table is the larger of `ColumnName` and the number of columns in `Data`.

`DeleteFcn`
string or function handle

Delete uitable callback routine. A callback routine that executes when you delete the `uitable` object (e.g., when you issue a `delete` command or clear the figure containing the `uitable`). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root CallbackObject property, which you can query using `gcbo`.

See "Function Handle Callbacks" for information on how to use function handles to define a callback function.

`Enable`
`{on} | inactive | off`

Enable or disable the uitable. This property determines how uitables respond to mouse button clicks, including which callback routines execute.

- `on` – The uitable is operational (the default).

- inactive – The uitable is not operational, but looks the same as when Enable is on.
- off – The uitable is not operational and its image is grayed out.

When you left-click on a uitable whose Enable property is on, MATLAB performs these actions in this order:

1. Sets the figure's SelectionType property.
2. Executes the uitable's CellSelectionCallback routine (but only for table cells, not header cells). Row and column indices of the cells the user selects continuously update the Indices field in the eventdata passed to the callback.
3. Does not set the figure's CurrentPoint property and does not execute either the table's ButtonDownFcn or the figure's WindowButtonDownFcn callback.

When you left-click on a uitable whose Enable property is off, or when you right-click a uitable whose Enable property has any value, MATLAB performs these actions in this order:

1. Sets the figure's SelectionType property.
2. Sets the figure's CurrentPoint property.
3. Executes the figure's WindowButtonDownFcn callback.

Extent

position rectangle (read only)

Size of uitable rectangle. A four-element vector of the form [0,0,width,height] that contains the calculated values of the largest extent of the table based on the current Data, RowName and ColumnName property values. Calculation depends on column and row widths, when they are available. The calculated extent can be larger than the figure.

The first two elements are always zero. `width` and `height` are the dimensions of the rectangle. All measurements are in units specified by the `Units` property.

When the uitable's `Units` property is set to `'Normalized'`, its `Extent` is measured relative to the figure, regardless of whether the table is contained in (parented to) a uipanel or not.

You can use this property to determine proper sizing for the uitable with respect to its content. Do this by setting the `width` and `height` of the uitable `Position` property to the width and height of the `Extent` property. However, doing this can cause the table to extend beyond the right or top edge of the figure and/or its uipanel parent, if any, for tables with large extents.

`FontAngle`

`{normal} | italic | oblique`

Character slant of cell content. MATLAB uses this property to select a font from those available on your particular system. Setting this property to `italic` or `oblique` selects a slanted version of the font, when it is available on your system.

`FontName`

string

Font family for cell content. The name of the font in which to display cell content. To display and print properly, this must be a font that your system supports. The default font is system dependent.

To use a fixed-width font that looks good in any locale (and displays properly in Japan, where multibyte character sets are used), set `FontName` to the string `FixedWidth` (this string value is case sensitive):

```
set(uitable_handle,'FontName','FixedWidth')
```

This parameter value eliminates the need to hard code the name of a fixed-width font, which may not display text properly on systems that do not use ASCII character encoding (such as in Japan). A properly written MATLAB application that needs to use a fixed-width font should set `FontName` to `FixedWidth` and rely on the root `FixedWidthFontName` property to be set correctly in the end user's environment.

End users can adapt a MATLAB application to different locales or personal environments by setting the root `FixedWidthFontName` property to the appropriate value for that locale from `startup.m`. Setting the root `FixedWidthFontName` property causes an immediate update of the display to use the new font.

`FontSize`
size in `FontUnits`

Font size for cell contents. A number specifying the size of the font in which to display cell contents, in units determined by the `FontUnits` property. The default point size is system dependent. If `FontUnits` is set to `normalized`, `FontSize` is a number between 0 and 1.

`FontUnits`
`{points} | normalized | inches |`
`centimeters | pixels`

Font size units for cell contents. This property determines the units used by the `FontSize` property. `Normalized` units interpret `FontSize` as a fraction of the height of the uitable. When you resize the uitable, MATLAB modifies the screen `FontSize` accordingly. `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = $^1/_{72}$ inch).

`FontWeight`
`light | {normal} | demi | bold`

Weight of cell text characters. MATLAB uses this property to select a font from those available on your particular system. Setting this property to `bold` causes MATLAB to use a bold version of the font, when it is available on your system.

`ForegroundColor`

`1-by-3 matrix of RGB triples or a color name`

Color of text in cells. Determines the color of the text defined for cell contents. Text in all cells share the current color. Specify as a 1-by-3 matrix of RGB triples, such as `[0 0 .8]` or as a color name. The default is a 1-by-3 matrix of platform-dependent colors. See `ColorSpec` for information about specifying RGB colors.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine

invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to `on` to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

`HitTest`
`{on} | off`

Selectable by mouse click. When `HitTest` is `off`, the `ButtonDownFcn` callback does not execute.

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For user interface objects, the `Interruptible` property affects the callbacks for these properties only:

- `ButtonDownFcn`
- `KeyPressFcn`
- `KeyReleaseFcn`
- `WindowButtonDownFcn`
- `WindowButtonMotionFcn`
- `WindowButtonUpFcn`
- `WindowKeyPressFcn`
- `WindowKeyReleaseFcn`

- `WindowScrollWheelFcn`

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

The `BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting the `Interruptible` property to `on` (default), allows a callback from other user interface objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted.

For more information, see "Control Callback Execution and Interruption".

Note If the interrupting callback is a `DeleteFcn`or `CreateFcn` callback, or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. An object's `ButtonDownFcn` or `Callback` routine is processed according to the rules described previously in this section.

`KeyPressFcn`

string or function handle

Key press callback function. A callback routine invoked by a key press when the callback's uitable object has focus. Focus is denoted by a border or a dotted border, respectively, in UNIX and Microsoft Windows. If no uitable has focus, the figure's key press callback function, if any, is invoked. `KeyPressFcn` can be a function handle, the name of a MATLAB function file, or any legal MATLAB expression.

If the specified value is the name of a function code file, the callback routine can query the figure's `CurrentCharacter` property to determine what particular key was pressed and thereby limit the callback execution to specific keys.

If the specified value is a function handle, the callback routine can retrieve information about the key that was pressed from its event data structure argument.

Event Data Structure Field	Description	Examples:			
		a	=	Shift	Shift/a
Character	Character interpretation of the key that was pressed.	'a'	'='	''	'A'
Modifier	Current modifier, such as 'control', or an empty cell array if there is no modifier	{1x0 cell}	{1x0 cell}	{'shift'}	{'shift'}
Key	Name of the key that was pressed.	'a'	'equal'	'shift'	'a'

The uitable KeyPressFcn callback executes for all keystrokes, including arrow keys or when a user edits cell content.

See "Function Handle Callbacks" for information on how to use function handles to define the callback function.

Parent
handle

Uitable parent. The handle of the uitable's parent object. You can move a uitable object to another figure, uipanel, or uibuttongroup by setting this property to the handle of the new parent.

Position
position rectangle

Size and location of uitable. The rectangle defined by this property specifies the size and location of the table within the parent figure window, ui, or uibuttongroup. Specify Position as a 4–element vector:

```
[left bottom width height]
```

left and bottom are the distance from the lower-left corner of the parent object to the lower-left corner of the uitable object.

`width` and `height` are the dimensions of the uitable rectangle. All measurements are in units specified by the `Units` property.

Note If you are specifying both `Units` and `Position` in the same call to `uitable`, specify `Units` first if you want `Position` to be interpreted using those units.

`RearrangeableColumns`

`on | {off}`

This object can be rearranged. The `RearrangeableColumns` property provides a mechanism that you can use to reorder the columns in the table. All columns are rearrangable when this property is turned on. MATLAB software sets the `RearrangeableColumns` property to `off` by default.

When this property is on, the user of a table can move any column of data (but not the row labels) at a time left or right to reorder it by clicking and dragging its header. Rearranging columns does not affect the ordering of columns in the table's `Data`, only the user's view of it.

`RowName`

1–by-n cell array of strings`|` {`'numbered'`} `|` empty matrix (`[]`)

Row heading names. Each element of the cell array is the name of a row. Row names are restricted to one line of text.

For sequentially numbered row headings starting with `1`, specify `RowName` as `'numbered'`. This is the default.

To remove the row headings, specify `RowName` as the empty matrix (`[]`).

The number of rows in the table is the larger of `RowName` and the number of rows in the `Data` property matrix or cell array.

`RowStriping`
: `{on} | off`

Color striping of table rows. When `RowStriping` is `on`, the background of consecutive rows of the table display in the pair of colors that the `BackgroundColor` color matrix specifies. The first color matrix row applies to odd-numbered rows, and the second to even-numbered rows. If the `BackgroundColor` matrix has only one row, it is applied to all rows (that is, no striping occurs).

When `RowStriping` is `off`, the first color specified for `BackgroundColor` is applied to all rows.

`Selected`
: `on | {off}`

Is object selected. When this property is `on`, MATLAB displays selection handles if the `SelectionHighlight` property is also `on`. You can, for example, define the `ButtonDownFcn` to set this property, allowing users to select the object with the mouse.

`SelectionHighlight`
: `{on} | off`

Object highlight when selected. When the `Selected` property is `on`, MATLAB indicates the selected state by drawing four edge handles and four corner handles. When `SelectionHighlight` is `off`, MATLAB does not draw the handles.

`Tag`
: string (GUIDE sets this property)

User-specified object label. The `Tag` property provides a means to identify graphics objects with a user-specified label. This is particularly useful when constructing interactive graphics programs that would otherwise need to define object handles as

global variables or pass them as arguments between callback routines. You can define `Tag` as any string.

`TooltipString`
string

Content of tooltip for object. The `TooltipString` property specifies the text of the tooltip associated with the uitable. When the user moves the mouse pointer over the table and leaves it there, the tooltip is displayed.

To create a tooltip that has more than one line of text, use `sprintf` to generate a string containing newline (`\n`) characters and then set the `TooltipString` to that value. For example:

```
h = uitable;
s = sprintf('UITable tooltip line 1\nUITable tooltip line 2');
set(h,'TooltipString',s)
```

`Type`
string (read only)

Class of graphics object. For uitable objects, `Type` is always the string `'uitable'`.

`UIContextMenu`
handle

Associate a context menu with uitable. Assign this property the handle of a `uicontextmenu` object. MATLAB displays the context menu whenever you right-click over the uitable. Use the `uicontextmenu` function to create the context menu.

`Units`
`{pixels} | normalized | inches | centimeters | points | characters` (GUIDE default: `normalized`)

Units of measurement. MATLAB uses these units to interpret the `Extent` and `Position` properties. All units are measured from the lower-left corner of the parent object.

- `Normalized` units map the lower-left corner of the parent object to (0,0) and the upper-right corner to (1.0,1.0).
- `pixels`, `inches`, `centimeters`, and `points` are absolute units (1 point = 1/72 inch).
- `Character` units are characters using the default system font; the width of one character is the width of the letter `x`, the height of one character is the distance between the baselines of two lines of text.

If you change the value of `Units`, it is good practice to return it to its default value after completing your computation so as not to affect other functions that assume `Units` is set to the default value.

`UserData`
matrix

User-specified data. Any data you want to associate with the uitable object. MATLAB does not use this data, but you can access it using `set` and `get`.

`Visible`
`{on} | off`

Uitable visibility. By default, all uitables are visible. When set to `off`, the uitable is not visible, but still exists and you can query and set its properties.

Note Setting `Visible` to `off` for uitables that are not displayed initially in the GUI, can result in faster startup time for the GUI.

---

uitoggletool

Purpose

Create toggle button on toolbar

Syntax

```
htt = uitoggletool
htt = uitoggletool('PropertyName1',value1,'PropertyName2',
   value2,...)
htt = uitoggletool(ht,...)
```

Description

`htt = uitoggletool` creates a toggle button on the uitoolbar at the top of the current figure window, sets all its properties to default values, and returns a handle to the tool. If no uitoolbar exists, one is created. The uitoolbar is the parent of the uitoggletool. Use the returned handle `htt` to set properties of the uitoggletool. The `OnCallback`, `OffCallback` and `ClickedCallback` use the handle as their first argument. The button has no icon, but its border highlights when you hover over it with the mouse cursor. Add an icon by setting `CData` for the tool. Type `get(htt)` to see a list of uitoggletool object properties and their current values. Type `set(htt)` to see a list of uitoggletool object properties you can set and legal property values.

`htt = uitoggletool('`*`PropertyName1`*`',value1,'`*`PropertyName2`*`',value2,...)` assigns the specified property values, and assigns default values to the remaining properties. You can change the property values at a later time using the `set` function. You can specify properties as parameter name/value pairs, cell arrays containing parameter names and values, or structures with fields containing parameter names and values as input arguments. For a complete list, see Uitoggletool Properties. Type `get(htt)` to see a list of uipushtool object properties and their current values. Type `set(htt)` to see a list of uipushtool object properties that you can set and their legal property values.

`htt = uitoggletool(ht,...)` creates a button with `ht` as a parent. `ht` must be a uitoolbar handle.

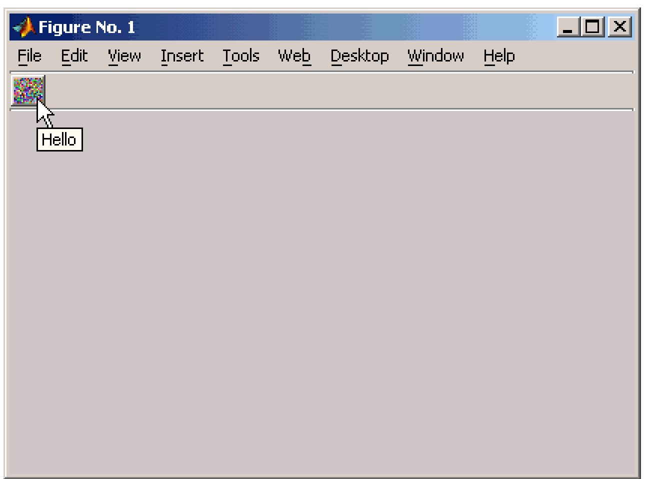
Toggle tools appear in figures whose `Window Style` is `normal` or `docked`. They do not appear in figures with a `'modal' WindowStyle`. If the `WindowStyle` property of a figure containing a tool bar and its toggle tool children changes to `modal`, the toggle tools continue to exist

as `Children` of the tool bar. The toggle tools do not display until you change the `WindowStyle` to `normal` or `docked`.

Examples

Create a `uitoolbar` object and places a `uitoggletool` object on it by specifying the toolbar handle as the toggle tool parent. Generate a random set of colors for the tool icon and specify a tool tip.

```
h = figure('ToolBar','none');
ht = uitoolbar(h);
a = rand(16,16,3);
htt = uitoggletool(ht,'CData',a,'TooltipString','Hello');
```



Alternatives

You can create toolbars with toggle tools using GUIDE.

See Also

`get` | `set` | `uicontrol` | `uipushtool` | `uitoolbar`

Tutorials

- “GUI That Accepts Property-Value Pairs”

How To

- “Create Toolbars for Programmatic GUIs”
- “Program Toolbar Tools”

Uitoggletool Properties property

Purpose

Describe toggle tool properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

You can set default Uitoggletool properties by typing:

```
set(h,'DefaultUitoggletoolPropertyName',PropertyValue...)
```

Where h can be the root handle (0), a figure handle, a uitoolbar handle, or a uitoggletool handle. *`PropertyName`* is the name of the Uitoggletool property and `PropertyValue` is the value you specify as the default for that property.

For more information about changing the default value of a property see "Setting Default Property Values".

Properties

This section lists all properties useful to `uitoggletool` objects along with valid values and a descriptions of their use. Curly braces { } enclose default values.

Property	Purpose
`BeingDeleted`	This object is being deleted.
`BusyAction`	Callback routine interruption.
`CData`	Truecolor image displayed on the toggle tool.
`ClickedCallback`	Control action independent of the toggle tool position.

Property	Purpose
`CreateFcn`	Callback routine executed during object creation.
`DeleteFcn`	Callback routine executed during object deletion.
`Enable`	Enable or disable the uitoggletool.
`HandleVisibility`	Control access to object's handle.
`HitTest`	Whether selectable by mouse click
`Interruptible`	Callback routine interruption mode.
`OffCallback`	Control action when toggle tool is set to the off position.
`OnCallback`	Control action when toggle tool is set to the on position.
`Parent`	Handle of uitoggletool's parent toolbar.
`Separator`	Separator line mode.
`State`	Uitoggletool state.
`Tag`	User-specified object label.
`TooltipString`	Content of object's tooltip.
`Type`	Object class.
`UIContextMenu`	Uicontextmenu object associated with the uitoggletool
`UserData`	User specified data.
`Visible`	Uitoggletool visibility.

`BeingDeleted`

`on | {off}` (read only)

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are

in the process of being deleted. MATLAB software sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property) It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, some functions may not need to perform actions on objects that are being deleted, and therefore, can check the object's `BeingDeleted` property before acting.

`BusyAction`

`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`CData`

3-dimensional array

Truecolor image displayed on control as its icon. An *n*-by-*m*-by-3 array of RGB values that defines a truecolor image displayed

on either a push button or toggle button. Each value must be between 0.0 and 1.0. If your `CData` array is larger than 16 in the first or second dimension, it can be clipped or result in other undesirable effects. If the array is clipped, only the center 16-by-16 part of the array is used.

`ClickedCallback`
string or function handle

Control action independent of the toggle tool position. A routine that executes after either the `OnCallback` routine or `OffCallback` routine runs to completion. The uitoggletool `Enable` property must be set to `on`.

`CreateFcn`
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a uitoggletool object. MATLAB sets all property values for the uitoggletool before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the toggle tool being created.

Setting this property on an existing uitoggletool object has no effect.

You can define a default `CreateFcn` callback for all new uitoggletools. This default applies unless you override it by specifying a different `CreateFcn` callback when you call `uitoggletool`. For example, the statement,

```
set(0,'DefaultUitoggletoolCreateFcn',...
    'set(gcbo,''Enable'',''off'')'
```

creates a default `CreateFcn` callback that runs whenever you create a new toggle tool. It sets the toggle tool `Enable` property to `off`.

To override this default and create a toggle tool whose `Enable` property is set to `on`, you could call `uitoggletool` with code similar to

```
htt = uitoggletool(...,'CreateFcn',...
                   'set(gcbo,''Enable'',''on'')',...)
```

Note To override a default `CreateFcn` callback you must provide a new callback and not just provide different values for the specified properties. This is because the `CreateFcn` callback runs after the property values are set, and can override property values you have set explicitly in the `uitoggletool` call. In the example above, if instead of redefining the `CreateFcn` property for this toggle tool, you had explicitly set `Enable` to `on`, the default `CreateFcn` callback would have set `CData` back to `off`.

Do not call `copyobj` or `textwrap` (which calls `copyobj`) inside a `CreateFcn`. The act of copying the uicontrol object fires the `CreateFcn` repeatedly, which raises a series of error messages after exceeding the root object's `RecursionLimit` property.

See Function Handle Callbacks for information on how to use function handles to define a callback function.

`DeleteFcn`
: string or function handle

Callback routine executed during object deletion. A callback routine that executes when you delete the uitoggletool object (e.g., when you call the `delete` function or cause the figure containing the uitoggletool to reset). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

The handle of the object whose `DeleteFcn` is being executed is accessible only through the root `CallbackObject` property, which you can query using `gcbo`.

See Function Handle Callbacks for information on how to use function handles to define a callback function.

`Enable`
{on} | off

Enable or disable the uitoggletool. This property controls how uitoggletools respond to mouse button clicks, including which callback routines execute.

- `on` – The uitoggletool is operational (the default).
- `off` – The uitoggletool is not operational and its icon (set by the `Cdata` property) is grayed out.

When you left-click on a uitoggletool whose `Enable` property is `on`, MATLAB performs these actions in this order:

1. Executes the toggle tool `OnCallback` or `OffCallback` routine, depending on its current state, and its `ClickedCallback` routine.
2. Does *not* set the figure `CurrentPoint` property and does *not* execute the figure's `WindowButtonDownFcn` callback.
3. Does *not* set the figure `SelectionType` property.

When you left-click a uitoggletool whose `Enable` property is `off`, or when you right-click a uitoggletool whose `Enable` property has any value, no action is reported, no callback executes, and neither the `SelectionType` nor `CurrentPoint` figure properties are modified.

`HandleVisibility`
`{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI (such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to on to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

`HitTest`
`{on} | off`

Selectable by mouse click. This property has no effect on uitoggletool objects.

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For user interface objects, the `Interruptible` property affects the callbacks for these properties only:

- `ButtonDownFcn`
- `KeyPressFcn`
- `KeyReleaseFcn`
- `WindowButtonDownFcn`
- `WindowButtonMotionFcn`
- `WindowButtonUpFcn`
- `WindowKeyPressFcn`
- `WindowKeyReleaseFcn`
- `WindowScrollWheelFcn`

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.

- If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

The `BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting the `Interruptible` property to `on` (default), allows a callback from other user interface objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted. For more information, see "Control Callback Execution and Interruption".

Note If the interrupting callback is a `DeleteFcn`or `CreateFcn` callback, or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. An object's `ButtonDownFcn` or `Callback` routine is processed according to the rules described previously in this section.

`OffCallback`
string or function handle

Control action. A routine that executes if the uitoggletool's `Enable` property is set to `on`, and either

- The toggle tool `State` is set to `off`.
- The toggle tool is set to the `off` position by pressing a mouse button while the pointer is on the toggle tool itself or in a 5-pixel wide border around it.

The `ClickedCallback` routine, if there is one, runs after the `OffCallback` routine runs to completion.

`OnCallback`
string or function handle

Control action. A routine that executes if the uitoggletool's `Enable` property is set to `on`, and either

- The toggle tool `State` is set to `on`.
- The toggle tool is set to the `on` position by pressing a mouse button while the pointer is on the toggle tool itself or in a 5-pixel wide border around it.

The `ClickedCallback` routine, if there is one, runs after the `OffCallback` routine runs to completion.

`Parent`
handle

Uitoggletool parent. The handle of the uitoggletool's parent toolbar. You can move a uitoggletool object to another toolbar by setting this property to the handle of the new parent.

`Separator`
`on | {off}`

Separator line mode. Setting this property to `on` draws a dividing line to left of the uitoggletool.

`State`

`on | {off}`

Uitoggletool state. When the state is `on`, the toggle tool appears in the down, or pressed, position. When the state is `off`, it appears in the up position. Changing the state causes the appropriate `OnCallback` or `OffCallback` routine to run.

`Tag`

string

User-specified object identifier. The `Tag` property provides a means to identify graphics objects with a user-specified label. You can define `Tag` as any string.

With the `findobj` function, you can locate an object with a given `Tag` property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified toolbars) that have the `Tag` value `'Bold'`.

```
h = findobj(uitoolbarhandles, 'Tag', 'Bold')
```

`TooltipString`

string

Content of tooltip for object. The `TooltipString` property specifies the text of the tooltip associated with the uitoggletool. When the user moves the mouse pointer over the control and leaves it there, the tooltip is displayed.

To create a tooltip that has more than one line of text, use `sprintf` to generate a string containing newline (`\n`) characters and then set the `TooltipString` to that value. For example:

```
h = uitoggletool;
s = sprintf('Toggletool tooltip line 1\nToggletool tooltip line 2');
set(h,'TooltipString',s)
```

`Type`
: `string` (read-only)

  Object class. This property identifies the kind of graphics object. For uitoggletool objects, `Type` is always the string `'uitoggletool'`.

`UIContextMenu`
: handle

  Associate a context menu with uicontrol. This property has no effect on uitoggletool objects.

`UserData`
: array

  User specified data. You can specify `UserData` as any array you want to associate with the uitoggletool object. The object does not use this data, but you can access it using the `set` and `get` functions.

`Visible`
: `{on} | off`

  Uitoggletool visibility. By default, all uitoggletools are visible. When set to `off`, the uitoggletool is not visible, but still exists and you can query and set its properties.

Purpose

Create toolbar on figure

Syntax

```
ht =
uitoolbar('PropertyName1',value1,'PropertyName2',value2,
   ...)
ht = uitoolbar(h,...)
```

Description

`ht =`
`uitoolbar('`*`PropertyName1`*`',value1,'`*`PropertyName2`*`',value2,...)` creates an empty toolbar at the top of the current figure window, and returns a handle to it. `uitoolbar` assigns the specified property values, and assigns default values to the remaining properties. You can change the property values at a later time using the `set` function.

Type `get(ht)` to see a list of `uitoolbar` object properties and their current values. Type `set(ht)` to see a list of `uitoolbar` object properties that you can set and legal property values. See the Uitoolbar Properties reference page for more information.

`ht = uitoolbar(h,...)` creates a toolbar with `h` as a parent. `h` must be a figure handle.

Tips

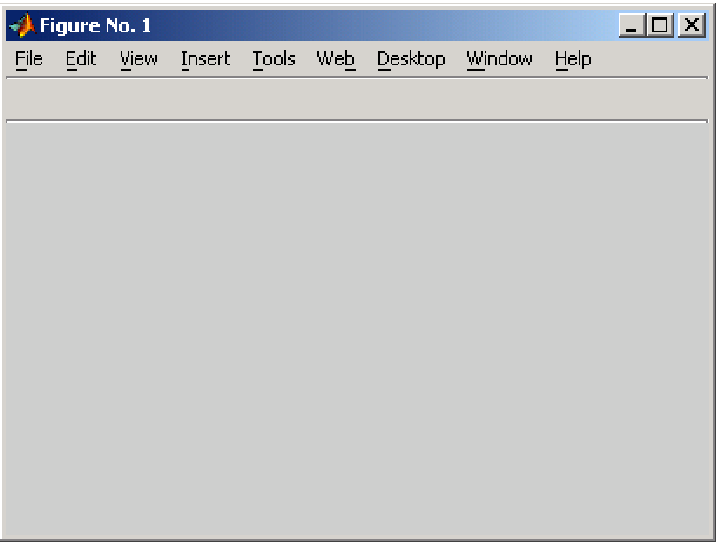
`uitoolbar` accepts property name/property value pairs, as well as structures and cell arrays of properties as input arguments.

Uitoolbars appear in figures whose `Window Style` is `normal` or `docked`. They do not appear in figures whose `WindowStyle` is `modal`. If the `WindowStyle` property of a figure containing a uitoolbar is changed to `modal`, the uitoolbar still exists and is contained in the `Children` list of the figure, but is not displayed until the `WindowStyle` is changed to `normal` or `docked`.

Examples

This example creates a figure with no toolbar, then adds a toolbar to it.

```
h = figure('ToolBar','none')
ht = uitoolbar(h)
```



See Also

`set` | `get` | `uicontrol` | `uipushtool` | `uitoggletool`

Uitoolbar Properties

Purpose

Describe toolbar properties

Modifying Properties

You can set and query graphics object properties in two ways:

- The Property Inspector is an interactive tool that enables you to see and change object property values. The Property inspector is available from GUIDE, or use the `inspect` function at the command line.
- The `set` and `get` functions enable you to set and query the values of properties.

You can set default Uitoolbar properties by typing:

```
set(h,'DefaultUitoolbarPropertyName',PropertyValue...)
```

Where h can be the root handle (0), a figure handle, or a uitoolbar handle. *`PropertyName`* is the name of the Uitoolbar property and `PropertyValue` is the value you specify as the default for that property.

For more information about changing the default value of a property see Setting Default Property Values.

Uitoolbar Properties

This section lists all properties useful to `uitoolbar` objects along with valid values and a descriptions of their use. Curly braces { } enclose default values.

Property	Purpose
`BeingDeleted`	This object is being deleted.
`BusyAction`	Callback routine interruption.
`Children`	Handles of uitoolbar's children.
`CreateFcn`	Callback routine executed during object creation.
`DeleteFcn`	Callback routine executed during object deletion.
`HandleVisibility`	Control access to object's handle.

Property	Purpose
HitTest	Whether selectable by mouse click
Interruptible	Callback routine interruption mode.
Parent	Handle of uitoolbar's parent.
Tag	User-specified object identifier.
Type	Object class.
UIContextMenu	Uicontextmenu object associated with the uitoolbar
UserData	User specified data.
Visible	Uitoolbar visibility.

BeingDeleted
`on | {off}` (read-only)

This object is being deleted. The `BeingDeleted` property provides a mechanism that you can use to determine if objects are in the process of being deleted. MATLAB software sets the `BeingDeleted` property to `on` when the object's delete function callback is called (see the `DeleteFcn` property) It remains set to `on` while the delete function executes, after which the object no longer exists.

For example, some functions may not need to perform actions on objects that are being deleted, and therefore, can check the object's `BeingDeleted` property before acting.

BusyAction
`cancel | {queue}`

Callback queuing

Determines how MATLAB handles the execution of interrupting callbacks.

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. The `BusyAction` property of the *interrupting* callback determines how MATLAB handles its execution. When the `BusyAction` property is set to:

- `'queue'` — Puts the *interrupting* callback in a queue to be processed after the *running* callback finishes execution.
- `'cancel'` — Discards the *interrupting* callback as MATLAB finishes execution.

For information about how the `Interruptible` property of the callback controls whether other callbacks can interrupt the *running* callback, see the `Interruptible` property description.

`Children`
vector of handles

Handles of tools on the toolbar. A vector containing the handles of all children of the uitoolbar object, in the order in which they appear on the toolbar. The children objects of uitoolbars are uipushtools and uitoggletools. You can use this property to reorder the children.

`CreateFcn`
string or function handle

Callback routine executed during object creation. The specified function executes when MATLAB creates a uitoolbar object. MATLAB sets all property values for the uitoolbar before executing the `CreateFcn` callback so these values are available to the callback. Within the function, use `gcbo` to get the handle of the toolbar being created.

Setting this property on an existing uitoolbar object has no effect.

You can define a default CreateFcn callback for all new uitoolbars. This default applies unless you override it by specifying a different CreateFcn callback when you call uitoolbar. For example, the statement,

```
set(0,'DefaultUitoolbarCreateFcn',...
    'set(gcbo,''Visibility'',''off'')')
```

creates a default CreateFcn callback that runs whenever you create a new toolbar. It sets the toolbar visibility to off.

To override this default and create a toolbar whose Visibility property is set to on, you could call uitoolbar with a call similar to

```
ht = uitoolbar(...,'CreateFcn',...
               'set(gcbo,''Visibility'',''on'')',...)
```

Note To override a default CreateFcn callback you must provide a new callback and not just provide different values for the specified properties. This is because the CreateFcn callback runs after the property values are set, and can override property values you have set explicitly in the uitoolbar call. In the example above, if instead of redefining the CreateFcn property for this toolbar, you had explicitly set Visibility to on, the default CreateFcn callback would have set Visibility back to off.

Do not call copyobj or textwrap (which calls copyobj) inside a CreateFcn. The act of copying the uicontrol object fires the CreateFcn repeatedly, which raises a series of error messages after exceeding the root object's RecursionLimit property.

See Function Handle Callbacks for information on how to use function handles to define a callback function.

`DeleteFcn`

string or function handle

Callback routine executed during object deletion. A callback function that executes when the uitoolbar object is deleted (e.g., when you call the `delete` function or cause the figure containing the uitoolbar to reset). MATLAB executes the routine before destroying the object's properties so these values are available to the callback routine.

Within the function, use `gcbo` to get the handle of the toolbar being deleted.

`HandleVisibility`

`{on} | callback | off`

Control access to object's handle. This property determines when an object's handle is visible in its parent's list of children. When a handle is not visible in its parent's list of children, it is not returned by functions that obtain handles by searching the object hierarchy or querying handle properties. This includes `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close`. Neither is the handle visible in the parent figure's `CurrentObject` property. Handles that are hidden are still valid. If you know an object's handle, you can `set` and `get` its properties, and pass it to any function that operates on handles.

- Handles are always visible when `HandleVisibility` is `on`.
- Setting `HandleVisibility` to `callback` causes handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This provides a means to protect GUIs from command-line users, while allowing callback routines to have complete access to object handles.
- Setting `HandleVisibility` to `off` makes handles invisible at all times. This may be necessary when a callback routine invokes a function that might potentially damage the GUI

(such as evaluating a user-typed string), and so temporarily hides its own handles during the execution of that function.

You can set the root `ShowHiddenHandles` property to on to make all handles visible, regardless of their `HandleVisibility` settings. This does not affect the values of the `HandleVisibility` properties.

`HitTest`
`{on} | off`

Selectable by mouse click. This property has no effect on uitoolbar objects.

`Interruptible`
`off | {on}`

Callback routine interruption

Controls whether MATLAB can interrupt an object's callback function when subsequent callbacks attempt to interrupt it.

For user interface objects, the `Interruptible` property affects the callbacks for these properties only:

- `ButtonDownFcn`
- `KeyPressFcn`
- `KeyReleaseFcn`
- `WindowButtonDownFcn`
- `WindowButtonMotionFcn`
- `WindowButtonUpFcn`
- `WindowKeyPressFcn`
- `WindowKeyReleaseFcn`
- `WindowScrollWheelFcn`

A *running* callback is the currently executing callback. The *interrupting* callback is the callback that tries to interrupt the *running* callback. MATLAB handles both callbacks based on the `Interruptible` property of the object of the *running* callback.

When the `Interruptible` property is set to:

- `'off'`, MATLAB finishes execution of the *running* callback without any interruptions
- `'on'`, these conditions apply:
  - If there is a `drawnow`, `figure`, `getframe`, `waitfor`, or `pause` command in the *running* callback, then MATLAB executes the *interrupting* callbacks which are already in the queue and returns to finish execution of the current callback.
  - If one of the above functions is not in the *running* callback, then MATLAB finishes execution of the current callback without any interruption.

The `BusyAction` property of the object of interrupting callback determines whether the callback should be ignored or should be put in the queue.

Setting the `Interruptible` property to `on` (default), allows a callback from other user interface objects to interrupt callback functions originating from this object.

Note MATLAB does not save the state of properties or the display when an interruption occurs. For example, the handle returned by the `gca` or `gcf` command may be changed as another callback is executed.

After the function that interrupts a callback completes, the callback resumes execution where it halted when interrupted.

For more information, see "Control Callback Execution and Interruption".

Note If the interrupting callback is a `DeleteFcn`or `CreateFcn` callback, or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of that object's `Interruptible` property. The interrupting callback starts execution at the next `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` statement. An object's `ButtonDownFcn` or `Callback` routine is processed according to the rules described previously in this section.

`Parent`
handle

Uitoolbar parent. The handle of the uitoolbar's parent figure. You can move a uitoolbar object to another figure by setting this property to the handle of the new parent.

`Tag`
string

User-specified object identifier. The `Tag` property provides a means to identify graphics objects with a user-specified label. You can define `Tag` as any string.

With the `findobj` function, you can locate an object with a given `Tag` property value. This saves you from defining object handles as global variables. For example, this function call returns the handles of all children (of the specified figures) that have the `Tag` value `'FormatTb'`.

```
h = findobj(figurehandles,'Tag','FormatTb')
```

`Type`
`string` (read-only)

Object class. This property identifies the kind of graphics object. For uitoolbar objects, `Type` is always the string `'uitoolbar'`.

`UIContextMenu`
handle

Associate a context menu with uicontrol. This property has no effect on uitoolbar objects.

`UserData`
array

User specified data. You can specify `UserData` as any array you want to associate with the uitoolbar object. The object does not use this data, but you can access it using the `set` and `get` functions.

`Visible`
`{on} | off`

Uitoolbar visibility. By default, all uitoolbars are visible. When set to `off`, the uitoolbar is not visible, but still exists and you can query and set its properties.

Purpose

Block program execution and wait to resume

Syntax

```
uiwait
uiwait(h)
uiwait(h,timeout)
```

Description

`uiwait` blocks execution until `uiresume` is called or the current figure is deleted. This syntax is the same as `uiwait(gcf)`.

`uiwait(h)` blocks execution until `uiresume` is called or the figure `h` is deleted.

`uiwait(h,timeout)` blocks execution until `uiresume` is called, the figure `h` is deleted, or `timeout` seconds elapse. The minimum value of `timeout` is 1. If `uiwait` receives a smaller value, it issues a warning and uses a 1 second `timeout`.

Tips

The `uiwait` and `uiresume` functions block and resume MATLAB and Simulink program execution. `uiwait` also blocks the execution of Simulink models. The functions `pause` (with no argument) and `waitfor` also block execution in this manner.`uiwait` is a convenient way to use the `waitfor` command. You typically use it in conjunction with a dialog box. It provides a way to block the execution of the MATLAB program that created the dialog, until the user responds to the dialog box. When used in conjunction with a modal dialog, `uiwait` can block the execution of the program file *and* restrict user interaction to the dialog only.

Examples

This example creates a GUI with a **Continue** push button. The example calls `uiwait` to block MATLAB execution until `uiresume` is called. This happens when the user clicks the **Continue** push button because the push button's `Callback`, which responds to the click, calls `uiresume`.

```
f = figure;
h = uicontrol('Position',[20 20 200 40],'String','Continue',...
              'Callback','uiresume(gcbf)');
disp('This will print immediately');
uiwait(gcf);
```

```
disp('This will print after you click Continue');
close(f);
```

`gcbf` is the handle of the figure that contains the object whose callback is executing.

See Also

`dialog` | `figure` | `uicontrol` | `uimenu` | `uiresume` | `waitfor`

Purpose Unary minus

Syntax

```
b = -a
b = uminus(a)
```

Description

`b = -a` negates the elements of array `a` and returns the result in array `b`.

`b = uminus(a)` is called for the syntax `-a` when `a` is an object.

See Also

`uplus`

undocheckout

Purpose

Undo previous checkout from source control system (UNIX platforms)

Syntax

```
undocheckout('filename')
undocheckout({'filename1','filename2', ...,'filenamen'})
```

Description

`undocheckout('filename')` makes the file `filename` available for checkout, where `filename` does not reflect any of the changes you made after you last checked it out. Use the full path for `filename` and include the file extension.

`undocheckout({'filename1','filename2', ...,'filenamen'})` makes `filename1` through `filenamen` available for checkout, where the files do not reflect any of the changes you made after you last checked them out. Use the full paths for the file names and include the file extensions.

Examples

Undo the checkouts of `/myserver/myfiles/clock.m` and `/myserver/myfiles/calendar.m` from the source control system:

```
undocheckout({'/myserver/myfiles/clock.m', ...
'/myserver/myfiles/calendar.m'})
```

See Also

`checkin` | `checkout` | `verctrl`

How To

- "Undo the Checkout (UNIX Platforms)"

Purpose

Convert Unicode characters to numeric bytes

Syntax

```
bytes = unicode2native(unicodestr)
bytes = unicode2native(unicodestr, encoding)
```

Description

`bytes = unicode2native(unicodestr)` takes a `char` vector of Unicode characters, `unicodestr`, converts it to the MATLAB default character encoding scheme, and returns the bytes as a `uint8` vector, `bytes`. Output vector `bytes` has the same general array shape as the `unicodestr` input. You can save the output of `unicode2native` to a file using the `fwrite` function.

`bytes = unicode2native(unicodestr, encoding)` converts the Unicode characters to the character encoding scheme specified by the string `encoding`. `encoding` must be the empty string (`''`) or a name or alias for an encoding scheme. Some examples are `'UTF-8'`, `'latin1'`, `'US-ASCII'`, and `'Shift_JIS'`. For common names and aliases, see the Web site `http://www.iana.org/assignments/character-sets`. If `encoding` is unspecified or is the empty string (`''`), the MATLAB default encoding scheme is used.

Examples

This example begins with two strings containing Unicode characters. It assumes that string `str1` contains text in a Western European language and string `str2` contains Japanese text. The example writes both strings into the same file, using the ISO-8859-1 character encoding scheme for the first string and the Shift-JIS encoding scheme for the second string. The example uses `unicode2native` to convert the two strings to the appropriate encoding schemes.

```
fid = fopen('mixed.txt', 'w');
bytes1 = unicode2native(str1, 'ISO-8859-1');
fwrite(fid, bytes1, 'uint8');
bytes2 = unicode2native(str2, 'Shift_JIS');
fwrite(fid, bytes2, 'uint8');
fclose(fid);
```

See Also

`native2unicode`

Purpose

Set union of two arrays

Syntax

```
C = union(A,B)
C = union(A,B,'rows')
[C,ia,ib] = union(A,B)
[C,ia,ib] = union(A,B,'rows')

[C,ia,ib] = union( ___ ,setOrder)

[C,ia,ib] = union(A,B,'legacy')

[C,ia,ib] = union(A,B,'rows','legacy')
```

Description

`C = union(A,B)` returns the combined values from `A` and `B` with no repetitions. The values of `C` are in sorted order.

`C = union(A,B,'rows')` treats each row of `A` and each row of `B` as single entities and returns the combined rows from `A` and `B` with no repetitions. The rows of the matrix `C` are in sorted order.

The `'rows'` option does not support cell arrays.

`[C,ia,ib] = union(A,B)` also returns index vectors `ia` and `ib`, such that the values in `C` are the combined values of `A(ia)` and `B(ib)`.

`[C,ia,ib] = union(A,B,'rows')` also returns index vectors `ia` and `ib` such that the rows of `C` are the combined rows of `A(ia,:)` and `B(ib,:)`.

`[C,ia,ib] = union( ___ ,setOrder)` returns `C` in a specific order using any of the input arguments in the previous syntaxes. `setOrder='sorted'` returns the values (or rows) of `C` in sorted order. `setOrder='stable'` returns the values (or rows) of `C` in the same order as `A` then `B`. If no value is specified, the default is `'sorted'`.

`[C,ia,ib] = union(A,B,'legacy')` and `[C,ia,ib] = union(A,B,'rows','legacy')` preserve the behavior of the `union` function from R2012b and prior releases.

Input Arguments

A,B - Input arrays

vectors | matrices | N-D arrays

Input arrays, specified as vectors, matrices, or N-D arrays whose elements can be any numeric, `logical`, or `char` data type. `A` and `B` also can be cell arrays of strings. Furthermore, `A` and `B` can be objects with the following class methods:

- `sort` (or `sortrows` for the `'rows'` option), where the second output of this method is `double` or another built-in numeric class
- `ne`

These objects include heterogeneous arrays derived from the same root class.

`A` and `B` must be of the same class with the following exceptions:

- `logical`, `char`, and all numeric classes can combine with `double` arrays.
- Cell arrays of strings can combine with `char` arrays.

If you specify the `'rows'` option, `A` and `B` must have the same number of columns.

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

setOrder - Order flag

`'sorted'` (default) | `'stable'`

Order flag, specified as `'sorted'` or `'stable'`, indicates the order of the values (or rows) in `C`.

Order Flag	Meaning
`'sorted'`	The values (or rows) in `C` return in sorted order. For example: `C = union([5 5 3],[1 2],'sorted')` returns `C = [1 2 3 5]`.
`'stable'`	The values (or rows) in `C` return in the same order as they appear in `A` and `B`. For example: `C = union([5 5 3],[1 2],'stable')` returns `C = [5 3 1 2]`.

Data Types
`char`

Output Arguments

`C` - Combined values of `A` and `B`

vector | matrix

Combined values of `A` and `B`, returned as a vector or matrix. The following describes the shape of `C` when the `'legacy'` flag is not specified:

- If the `'rows'` flag is not specified, then `C` is a column vector unless both `A` and `B` are row vectors.
- If the `'rows'` flag is not specified and both `A` and `B` are row vectors, then `C` is a row vector.
- If the `'rows'` flag is specified, then `C` is a matrix containing the combined rows of `A` and `B`.

The class of the inputs `A` and `B` determines the class of `C`:

- If the class of `A` and `B` are the same, then `C` is the same class.
- If you combine a `char` or nondouble numeric class with `double`, then `C` is the same class as the nondouble input.
- If you combine a `logical` class with `double`, then `C` is `double`.

- If you combine a cell array of strings with `char`, then `C` is a cell array of strings.

ia - Index to A

column vector

Index to `A`, returned as a column vector when the `'legacy'` flag is not specified. `ia` indicates the values (or rows) in `A` that contribute to the union. If a value (or row) appears multiple times in `A`, then `ia` contains the index to the first occurrence of the value (or row). If a value appears in both `A` and `B`, then `ia` contains the index to the first occurrence in `A`.

ib - Index to B

column vector

Index to `B`, returned as a column vector when the `'legacy'` flag is not specified. `ib` indicates the values (or rows) in `B` that contribute to the union. If there is a repeated value (or row) appearing exclusively in `B`, then `ib` contains the index to the first occurrence of the value. If a value (or row) appears in both `A` and `B`, then `ib` does not contain an index to the value (or row).

Examples

Union of Two Vectors

Define two vectors with a value in common.

```
A = [5 7 1]; B = [3 1 1];
```

Find the union of vectors `A` and `B`.

```
C = union(A,B)
```

```
C =

     1     3     5     7
```

Union of Two Vectors and Their Indices

Define two vectors with a value in common.

```
A = [5 7 1]; B = [3 1 1];
```

Find the union of vectors `A` and `B` as well as the index vectors `ia` and `ib`.

```
[C,ia,ib] = union(A,B)

C =

     1     3     5     7


ia =

     3
     1
     2


ib =

     1
```

The values in `C` are the combined values of `A(ia)` and `B(ib)`.

Union of the Rows in Two Matrices

Define two matrices with a row in common.

```
A = [2 2 2; 0 0 1];
B = [1 2 3; 2 2 2; 2 2 2];
```

Find the combined rows of `A` and `B`, with no repetition, as well as the index vectors `ia` and `ib`.

```
[C,ia,ib] = union(A,B,'rows')

C =
```

```
     0     0     1
     1     2     3
     2     2     2


ia =

     2
     1


ib =

     1
```

The rows of `C` are the combined rows of `A(ia,:)` and `B(ib,:)`.

Union of Two Vectors with Specified Output Order

Use the `setOrder` argument to specify the ordering of the values in `C`.

Specify `'stable'` if you want the values in `C` to have the same order as in `A` and `B`.

```
A = [5 7 1]; B = [3 1 1];
[C,ia,ib] = union(A,B,'stable')
```

```
C =

     5     7     1     3


ia =

     1
     2
     3
```

```
ib =

     1
```

Alternatively, you can specify 'sorted' order.

```
A = [5 7 1]; B = [3 1 1];
[C,ia,ib] = union(A,B,'sorted')
```

```
C =

     1     3     5     7


ia =

     3
     1
     2


ib =

     1
```

Union of Vectors Containing NaNs

Define two vectors containing NaN.

```
A = [5 NaN 1]; B = [4 NaN NaN];
```

Find the union of vectors A and B.

```
C = union(A,B)
```

```
C =

     1     4     5   NaN   NaN   NaN
```

`union` treats `NaN` values as distinct.

Cell Array of Strings with Trailing White Space

Create a cell array of strings, `A`.

```
A = {'dog','cat','fish','horse'};
```

Create a cell array of strings, `B`, where some of the strings have trailing white space.

```
B = {'dog ','cat','fish ','horse'};
```

Combine the elements of `A` and `B`.

```
[C,ia,ib] = union(A,B)
```

```
C =

    'cat'    'dog'    'dog '    'fish'    'fish '    'horse'


ia =

     2
     1
     3
     4


ib =

     1
     3
```

`union` treats trailing white space in cell arrays of strings as distinct characters.

Union of Vectors of Different Classes and Shapes

Create a column vector character array.

```
A = ['A';'B';'C'], class(A)
```

```
A =

A
B
C


ans =

char
```

Create a row vector containing elements of numeric type `double`.

```
B = [68 69 70], class(B)
```

```
B =

    68    69    70


ans =

double
```

The `union` of `A` and `B` returns a column vector character array.

```
C = union(A,B), class(C)
```

```
C =
```

```
A
B
C
D
E
F


ans =

char
```

Union of Char and Cell Array of Strings

Create a character array containing the letters `a` , `b`, and `c`.

```
A = ['a';'b';'c'];
class(A)

ans =

char
```

Create a cell array of strings containing the letters `c`, `d`, and `e`.

```
B = {'c','d','e'};
class(B)

ans =

cell
```

Combine the elements of `A` and `B`.

```
C = union(A,B)
```

```
C = 

    'a'
    'b'
    'c'
    'd'
    'e'
```

The result, `C`, is a cell array of strings.

```
class(C)

ans =

cell
```

Preserve Legacy Behavior of union

Use the `'legacy'` flag to preserve the behavior of `union` from R2012b and prior releases in your code.

Find the union of `A` and `B` with the current behavior.

```
A = [5 7 1]; B = [3 1 1];
[C1,ia1,ib1] = union(A,B)

C1 =

     1     3     5     7


ia1 =

     3
     1
     2
```

```
ib1 =

     1
```

Find the union of A and B, and preserve the legacy behavior.

```
A = [5 7 1]; B = [3 1 1];
[C2,ia2,ib2] = union(A,B,'legacy')

C2 =

     1     3     5     7


ia2 =

     1     2


ib2 =

     3     1
```

See Also

unique | intersect | ismember | issorted | setdiff | setxor | sort

unique

Purpose

Unique values in array

Syntax

```
C = unique(A)
C = unique(A,'rows')
[C,ia,ic] = unique(A)
[C,ia,ic] = unique(A,'rows')

[C,ia,ic] = unique(A,setOrder)

[C,ia,ic] = unique(A,'rows',setOrder)

[C,ia,ic] = unique(A,'legacy')

[C,ia,ic] = unique(A,'rows','legacy')

[C,ia,ic] = unique(A,occurrence,'legacy')

[C,ia,ic] = unique(A,'rows',occurrence,'legacy')
```

Description

`C = unique(A)` returns the same values as in `A`, but with no repetitions. The values of `C` are in sorted order.

`C = unique(A,'rows')` treats each row of `A` as a single entity and returns the unique rows of `A`. The rows of the matrix `C` are in sorted order.

The `'rows'` option does not support cell arrays.

`[C,ia,ic] = unique(A)` also returns index vectors `ia` and `ic`, such that `C = A(ia)` and `A = C(ic)`.

`[C,ia,ic] = unique(A,'rows')` also returns index vectors `ia` and `ic`, such that `C = A(ia,:)` and `A = C(ic,:)`.

[C,ia,ic] = unique(A,setOrder) and [C,ia,ic] = unique(A,'rows',setOrder) return C in a specific order. setOrder='sorted' returns the values (or rows) of C in sorted order. setOrder='stable' returns the values (or rows) of C in the same order as A. If no value is specified, the default is 'sorted'.

[C,ia,ic] = unique(A,'legacy'), [C,ia,ic] = unique(A,'rows','legacy'), [C,ia,ic] = unique(A,occurrence,'legacy'),and [C,ia,ic] = unique(A,'rows',occurrence,'legacy') preserve the behavior of the unique function from R2012b and prior releases.

Input Arguments

A - Input array

vector | matrix | N-D array

Input array, specified as a vector, matrix, or N-D array whose elements can be any numeric, logical, or char data type. A also can be a cell array of strings. Furthermore, A can be an object with the following class methods:

- sort (or sortrows for the 'rows' option), where the second output of this method is double or another built-in numeric class
- ne

These objects include heterogeneous arrays derived from the same root class.

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

setOrder - Order flag

'sorted' (default) | 'stable'

Order flag, specified as 'sorted' or 'stable', indicates the order of the values (or rows) in C.

Order Flag	Meaning
'sorted'	The values (or rows) in C return in sorted order. For example: C = unique([5 5 3 4],'sorted') returns C = [3 4 5].
'stable'	The values (or rows) in C return in the same order as in A. For example: C = unique([5 5 3 4],'stable') returns C = [5 3 4].

Data Types
char

occurrence - Occurrence flag for legacy behavior

'last' (default) | 'first'

Occurrence flag for legacy behavior, specified as 'first' or 'last', indicates whether ia should contain the first or last indices to repeated values found in A.

Occurrence Flag	Meaning
'last'	If there are repeated values (or rows) in A, then ia contains the index to the last occurrence of the repeated value. For example: [C,ia,ic] = unique([9 9 9],'last','legacy') return ia = 3.
'first'	If there are repeated values (or rows) in A, then ia contains the index to the first occurrence of the repeated value. For example: [C,ia,ic] = unique([9 9 9],'first','legacy') retur ia = 1.

Data Types
`char`

Output Arguments

`C` - Unique values of `A`

vector | matrix

Unique values of `A`, returned as a vector or matrix. The following describes the shape of `C`:

- If the `'rows'` flag is not specified and `A` is a row vector, then `C` is a row vector.
- If the `'rows'` flag is not specified and `A` is not a row vector, then `C` is a column vector.
- If the `'rows'` flag is specified, then `C` is a matrix containing the unique rows of `A`.

The class of `C` is the same as the class of the input `A`.

ia - Index to `A`

column vector

Index to `A`, returned as a column vector when the `'legacy'` flag is not specified. `ia` contains indices, such that `C = A(ia)` (or `C = A(ia,:)` for the `'rows'` option).

ic - Index to `C`

column vector

Index to `C`, returned as a column vector when the `'legacy'` flag is not specified. `ic` contains indices, such that `A = C(ic)` (or `A = C(ic,:)` for the `'rows'` option).

Examples

Unique Values in a Vector

Define a vector with a repeated value.

```
A = [9 2 9 5];
```

Find the unique values of A.

```
C = unique(A)
```

```
C =

     2     5     9
```

Unique Values and their Indices

Define a vector with a repeated value.

```
A = [9 2 9 5];
```

Find the unique values of A and the index vectors ia and ic, such that C = A(ia) and A = C(ic).

```
[C, ia, ic] = unique(A)
```

```
C =

     2     5     9


ia =

     2
     4
     1


ic =

     3
     1
     3
     2
```

Unique Rows in a Matrix

Define a matrix with a repeated row.

```
A = [9 2 9 5; 9 2 9 0; 9 2 9 5];
```

Find the unique rows of `A` and the index vectors `ia` and `ic`, such that `C = A(ia,:)` and `A = C(ic,:)`.

```
[C, ia, ic] = unique(A,'rows')

C =

     9     2     9     0
     9     2     9     5


ia =

     2
     1


ic =

     2
     1
     2
```

Notice that the second row in `A` is identified as unique even though the elements `9`, `2`, and `9` are repeated in the other rows.

Unique Values in a Vector with Specified Output Order

Use the `setOrder` argument to specify the ordering of the values in `C`.

Specify `'stable'` if you want the values in `C` to have the same order as in `A`.

```
A = [9 2 9 5];
[C, ia, ic] = unique(A,'stable')

C =

     9     2     5


ia =

     1
     2
     4


ic =

     1
     2
     1
     3
```

Alternatively, you can specify `'sorted'` order.

```
[C, ia, ic] = unique(A,'sorted')

C =

     2     5     9


ia =

     2
     4
     1
```

```
ic =

     3
     1
     3
     2
```

Unique Values in Array Containing NaNs

Define a vector containing NaN.

```
A = [5 5 NaN NaN];
```

Find the unique values of A.

```
C = unique(A)
```

```
C =

     5   NaN   NaN
```

unique treats NaN values as distinct.

Unique Entries in Cell Array of Strings

Define a cell array of strings.

```
A = {'one','two','twenty-two','One','two'};
```

Find the unique strings contained in A.

```
C = unique(A)
```

```
C =

    'One'    'one'    'twenty-two'    'two'
```

Cell Array of Strings with Trailing White Space

Define a cell array of strings, A, where some of the strings have trailing white space.

```
A = {'dog','cat','fish','horse','dog ','fish '};
```

Find the unique strings contained in A.

```
C = unique(A)
```

```
C =

    'cat'    'dog'    'dog '    'fish'    'fish '    'horse'
```

unique treats trailing white space in cell arrays of strings as distinct characters.

Preserve Legacy Behavior of unique

Use the 'legacy' flag to preserve the behavior of unique from R2012b and prior releases in your code.

Find the unique elements of A with the current behavior.

```
A = [9 2 9 5];
[C1, ia1, ic1] = unique(A)
```

```
C1 =

     2     5     9


ia1 =

     2
     4
     1
```

```
ic1 =

     3
     1
     3
     2
```

Find the unique elements of A, and preserve the legacy behavior.

```
[C2, ia2, ic2] = unique(A, 'legacy')
```

```
C2 =

     2     5     9


ia2 =

     2     4     3


ic2 =

     3     1     3     2
```

See Also

union | intersect | ismember | issorted | setdiff | setxor | sort

Purpose

Execute UNIX command and return output

Syntax

```
status = unix(command)
[status,cmdout] = unix(command)
[status,cmdout] = unix(command,'-echo')
```

Description

`status = unix(command)` calls the UNIX operating system to execute the specified command. The operation waits for the command to finish execution before returning the exit status of the command to the `status` variable.

`[status,cmdout] = unix(command)` additionally returns the standard output of the command to `cmdout`. This syntax is most useful for commands that do not require user input.

`[status,cmdout] = unix(command,'-echo')` additionally displays (echoes) the command output in the MATLAB Command Window. This syntax is most useful for commands that require user input and that run correctly in the MATLAB Command Window.

Input Arguments

command - UNIX command

string

UNIX command, specified as a string. The `command` executes in a UNIX shell, which might not be the shell from which you launched MATLAB.

Example: `'ls'`

Output Arguments

status - Command exit status

0 | nonzero integer

Command exit status, returned as either `0` or a nonzero integer. When the command is successful, `status` is `0`. Otherwise, `status` is a nonzero integer.

- If `command` includes the ampersand character (&), then `status` is the exit status upon command launch.

- If `command` does not include the ampersand character (`&`), then `status` is the exit status upon command completion.

cmdout - Output of operating system command
string

Output of the operating system command, returned as a string.

Examples

Save UNIX Command Exit Status and Output

List all users who are currently logged in, and save the command exit status and output. Then, view the status.

```
command = 'who';
[status,cmdout] = unix(command);
status
```

```
status =

     0
```

A `status` of zero indicates that the command completed successfully. MATLAB returns a string containing the list of users in `cmdout`.

Tips

- To execute the operating system command in the background, include the trailing character, `&`, in the `command` argument (for example, `'emacs &'`). The exit status is immediately returned to the `status` variable. This syntax is useful for console programs that require interactive user command input while they run, and that do not run correctly in the MATLAB Command Window.

  Note If `command` includes the trailing `&` character, `cmdout` is empty.

- The `unix` function redirects `stdin` to the invoked command, `command`, by default. This redirection also forwards MATLAB script commands and the keyboard type-ahead buffer to the invoked command while

the `unix` function executes. This can lead to corrupted output when `unix` does not complete execution immediately. To disable `stdin` and type-ahead redirection, include the formatted string `< /dev/null` in the call to the invoked command.

Algorithms

MATLAB uses a shell program to execute the given command. It determines which shell program to use by checking environment variables on your system. MATLAB first checks the `MATLAB_SHELL` variable, and if either empty or not defined, then checks `SHELL`. If `SHELL` is also empty or not defined, MATLAB uses `/bin/sh`.

See Also

`!` (exclamation point) | `computer` | `dos` | `perl` | `system`

Concepts

- "Running External Commands, Scripts, and Programs"

Purpose

Unload shared library from memory

Syntax

```
unloadlibrary libname
```

Description

`unloadlibrary libname` unloads library `libname` from memory.

Limitations

- Use with libraries that are loaded using the `loadlibrary` function.

Input Arguments

libname - Name of shared library

string

Name of shared library, specified as a string. If you call `loadlibrary` using the `alias` option, then you must use the alias name for the `libname` argument.

Data Types
`char`

Examples

Unload Library

Add path to examples folder.

```
addpath(fullfile(matlabroot,'extern','examples','shrlib'))
```

Load the library, if it is not already loaded.

```
if ~libisloaded('shrlibsample')
   loadlibrary('shrlibsample')
end
```

Clean up.

```
unloadlibrary shrlibsample
```

Tips

- To unload a MEX-file, use the `clear` function.

See Also

`loadlibrary` | `libisloaded` | `clear`

unmesh

Purpose

Convert edge matrix to coordinate and Laplacian matrices

Syntax

```
[L,XY] = unmesh(E)
```

Description

`[L,XY] = unmesh(E)` returns the Laplacian matrix `L` and mesh vertex coordinate matrix `XY` for the `M`-by-4 edge matrix `E`. Each row of the edge matrix must contain the coordinates `[x1 y1 x2 y2]` of the edge endpoints.

Input Arguments

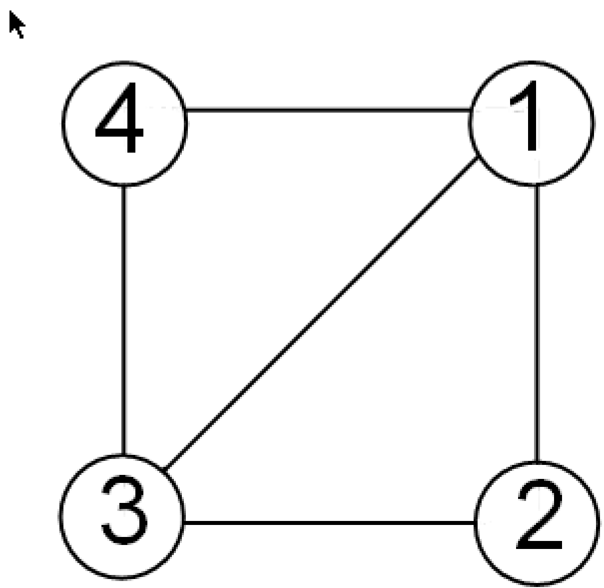
`E`	M-by-4 edge matrix `E`.

Output Arguments

`L`	Laplacian matrix representation of the graph.
`XY`	Mesh vertex coordinate matrix.

Examples

Take a simple example of a square with vertices at (1,1), (1,–1),(–1,–1), and (–1,1), where the connections between vertices are the four perpendicular edges of the square plus one diagonal connection between (–1, –1) and (1,1).



The edge matrix E for this graph is:

```
E=[1 1 1 -1;  % edge from 1 to 2
1 -1 -1 -1;   % edge from 2 to 3
-1 -1 -1 1;   % edge from 3 to 4
-1 -1 1 1;    % edge from 4 to 1
-1 1 1 1]     % edge from 3 to 1
```

Use `unmesh` to create the output matrices,

```
[A,XY]=unmesh(E);
4 vertices:
4/4
```

The Laplacian matrix is defined as

$$L_{ij} = \begin{cases} \deg(v_i) & \text{if } i = j \\ -1 & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases}$$

`unmesh` returns the Laplacian matrix `L` in sparse notation.

```
L

L =

   (1,1)        3
   (2,1)       -1
   (3,1)       -1
   (4,1)       -1
   (1,2)       -1
   (2,2)        2
   (4,2)       -1
   (1,3)       -1
   (3,3)        2
   (4,3)       -1
   (1,4)       -1
   (2,4)       -1
   (3,4)       -1
```

To see `L` in regular matrix notation, use the `full` command.

```
full(L)

ans =

     3    -1    -1    -1
    -1     2     0    -1
    -1     0     2    -1
    -1    -1    -1     3
```

The mesh coordinate matrix `XY` returns the coordinates of the corners of the square.

```
XY

XY =

    -1    -1
    -1     1
     1    -1
     1     1
```

See Also

gplot | treeplot

unmkpp

Purpose

Piecewise polynomial details

Syntax

```
[breaks,coefs,l,k,d] = unmkpp(pp)
```

Description

`[breaks,coefs,l,k,d] = unmkpp(pp)` extracts, from the piecewise polynomial `pp`, its breaks `breaks`, coefficients `coefs`, number of pieces `l`, order `k`, and dimension `d` of its target. Create `pp` using `spline` or the spline utility `mkpp`.

Examples

This example creates a description of the quadratic polynomial

$$\frac{-x^2}{4} + x$$

as a piecewise polynomial `pp`, then extracts the details of that description.

```
pp = mkpp([-8 -4],[-1/4 1 0]);
[breaks,coefs,l,k,d] = unmkpp(pp)

breaks =
    -8    -4

coefs =
   -0.2500    1.0000         0

l =
     1

k =
     3

d =
     1
```

See Also

`mkpp` | `ppval` | `spline`

Purpose

Unregister all event handlers associated with COM object events at run time

Syntax

```
h.unregisterallevents
unregisterallevents(h)
```

Description

`h.unregisterallevents` unregisters all events previously registered with COM object `h`. After calling `unregisterallevents`, the object no longer responds to any events until you register them again using the `registerevent` function.

`unregisterallevents(h)` is an alternate syntax.

COM functions are available on Microsoft Windows systems only.

Examples

Register and unregister events for an instance of the `mwsamp` control, using the `eventlisteners` function to see the event handler associated with each event:

1. Register three events and their respective handler routines.

   ```
   f = figure ('position', [100 200 200 200]);
   h = actxcontrol('mwsamp.mwsampctrl.2', ...
       [0 0 200 200], f, ...
       {'Click' 'myclick'; 'DblClick' 'my2click'; ...
       'MouseDown' 'mymoused'});
   h.eventlisteners
   ```

   MATLAB displays:

   ```
   ans =
       'click'        'myclick'
       'dblclick'     'my2click'
       'mousedown'    'mymoused'
   ```

2. Unregister all events simultaneously with `unregisterallevents`. `eventlisteners` returns an empty cell array, indicating that there are no longer any events registered with the control:

```
h.unregisterallevents;
h.eventlisteners
```

MATLAB displays:

```
ans =
     {}
```

See Also

events (COM) | eventlisteners | registerevent | unregisterevent | isevent

Purpose

Unregister event handler associated with COM object event at run time

Syntax

```
h.unregisterevent(eventhandler)
unregisterevent(h, eventhandler)
```

Description

`h.unregisterevent(eventhandler)` unregisters specific event handler routines from their corresponding events. Once you unregister an event, the object no longer responds to the event.

`unregisterevent(h, eventhandler)` is an alternate syntax.

You can unregister events at any time after creating a control. The `eventhandler` argument, which is a cell array, specifies both events and event handlers.

```
h.unregisterevent({'event_name',@event_handler});
```

Specify events in the `eventhandler` argument using the names of the events. Strings used in the `eventhandler` argument are not case sensitive. `unregisterevent` does not accept numeric event identifiers.

COM functions are available on Microsoft Windows systems only.

Examples

Unregister events for a control:

1. Create an `mwsamp` control and register all events with the same handler routine, `sampev`. Use `eventlisteners` to see the event handler used by each event. In this case, each event, when fired, calls `sampev.m`:

   ```
   f = figure ('position', [100 200 200 200]);
   h = actxcontrol('mwsamp.mwsampctrl.2', ...
       [0 0 200 200], f, ...
       'sampev');
   h.eventlisteners
   ```

   MATLAB displays:

```
ans =
    'Click'          'sampev'
    'DblClick'       'sampev'
    'MouseDown'      'sampev'
    'Event_Args'     'sampev'
```

2 Unregister just the `dblclick` event. Now, when you list the registered events using `eventlisteners`, `dblclick` is no longer registered and the control does not respond when you double-click the mouse over it:

```
h.unregisterevent({'dblclick' 'sampev'});
h.eventlisteners
```

MATLAB displays:

```
ans =
    'Click'          'sampev'
    'MouseDown'      'sampev'
    'Event_Args'     'sampev'
```

3 Now, register the `click` and `dblclick` events with a different event handler for `myclick` and `my2click`, respectively:

```
h.unregisterallevents;
h.registerevent({'click' 'myclick'; ...
    'dblclick' 'my2click'});
h.eventlisteners
```

MATLAB displays:

```
ans =
    'click'        'myclick'
    'dblclick'     'my2click'
```

4 Unregister these same events by specifying event names and their handler routines in a cell array. `eventlisteners` now returns an

empty cell array, meaning that no events are registered for the `mwsamp` control:

```
h.unregisterevent({'click' 'myclick'; ...
    'dblclick' 'my2click'});
h.eventlisteners
```

MATLAB displays:

```
ans =
     {}
```

Unregister Microsoft Excel workbook events:

1. Create a `Workbook` object and register two events with the event handler routines, `EvtActivateHndlr` and `EvtDeactivateHndlr`:

   ```
   myApp = actxserver('Excel.Application');
   wbs = myApp.Workbooks;
   wb = wbs.Add;wb.registerevent({'Activate' 'EvtActivateHndlr'; ...
       'Deactivate' 'EvtDeactivateHndlr'})
   wb.eventlisteners
   ```

   MATLAB shows the events with the corresponding event handlers.

   ```
   ans =
       'Activate'      'EvtActivateHndlr'
       'Deactivate'    'EvtDeactivateHndlr'
   ```

2. Next, unregister the `Deactivate` event handler:

   ```
   wb.unregisterevent({'Deactivate' 'EvtDeactivateHndlr'})
   wb.eventlisteners
   ```

   MATLAB shows the remaining registered event (`Activate`) with its corresponding event handler.

```
ans =
    'Activate'    'EvtActivateHndlr'
```

See Also

`events (COM)` | `eventlisteners` | `registerevent` | `unregisterallevents` | `isevent`

How To

- "Writing Event Handlers"

Purpose

Extract contents of tar file

Syntax

```
untar(tarfilename)
untar(tarfilename,outputdir)
untar(url, ...)
filenames = untar(...)
```

Description

`untar(tarfilename)` extracts the archived contents of `tarfilename` into the current directory and sets the files' attributes. It overwrites any existing files with the same names as those in the archive if the existing files' attributes and ownerships permit it. For example, if you rerun `untar` on the same `tarfilename`, MATLAB software does not overwrite files with a read-only attribute; instead, `untar` displays a warning for such files. On Microsoft Windows platforms, the hidden, system, and archive attributes are not set.

`tarfilename` is a string specifying the name of the tar file. `tarfilename` is gunzipped to a temporary directory and deleted if its extension ends in `.tgz` or `.gz`. If an extension is omitted, `untar` searches for `tarfilename` appended with `.tgz`, `.tar.gz`, or `.tar`. `tarfilename` can include the directory name; otherwise, the file must be in the current directory or in a directory on the MATLAB path.

`untar(tarfilename,outputdir)` uncompresses the archive `tarfilename` into the directory `outputdir`. If `outputdir` does not exist, MATLAB creates it.

`untar(url, ...)` extracts the tar archive from an Internet URL. The URL must include the protocol type (for example, `'http://'` or `'ftp://'`). MATLAB downloads the URL is to a temporary directory, and then deletes it.

`filenames = untar(...)` extracts the tar archive and returns the names of the extracted files in the string cell array `filenames`. If `outputdir` specifies a relative path, `filenames` contains the relative path. If `outputdir` specifies an absolute path, `filenames` contains the absolute path.

Examples

Using tar and untar to Copy Files

Copy all `.m` files in the current directory to the directory `backup`.

```
tar('mymfiles.tar.gz','*.m');
untar('mymfiles','backup');
```

Using untar with URL

Run `untar` to list Cleve Moler's "Numerical Computing with MATLAB" examples to the output directory `ncm`.

```
url ='http://www.mathworks.com/moler/ncm.tar.gz';
ncmFiles = untar(url,'ncm')
```

See Also

`gzip` | `gunzip` | `tar` | `unzip` | `zip`

Purpose

Correct phase angles to produce smoother phase plots

Syntax

```
Q = unwrap(P)
Q = unwrap(P,tol)
Q = unwrap(P,[],dim)
Q = unwrap(P,tol,dim)
```

Description

`Q = unwrap(P)` corrects the radian phase angles in a vector `P` by adding multiples of $\pm 2\pi$ when absolute jumps between consecutive elements of `P` are greater than or equal to the default jump tolerance of π radians. If `P` is a matrix, `unwrap` operates columnwise. If `P` is a multidimensional array, `unwrap` operates on the first nonsingleton dimension.

`Q = unwrap(P,tol)` uses a jump tolerance `tol` instead of the default value, π.

`Q = unwrap(P,[],dim)` unwraps along `dim` using the default tolerance.

`Q = unwrap(P,tol,dim)` uses a jump tolerance of `tol`.

Note A jump tolerance less than π has the same effect as a tolerance of π. For a tolerance less than π, if a jump is greater than the tolerance but less than π, adding $\pm 2\pi$ would result in a jump larger than the existing one, so `unwrap` chooses the current point. If you want to eliminate jumps that are less than π, try using a finer grid in the domain.

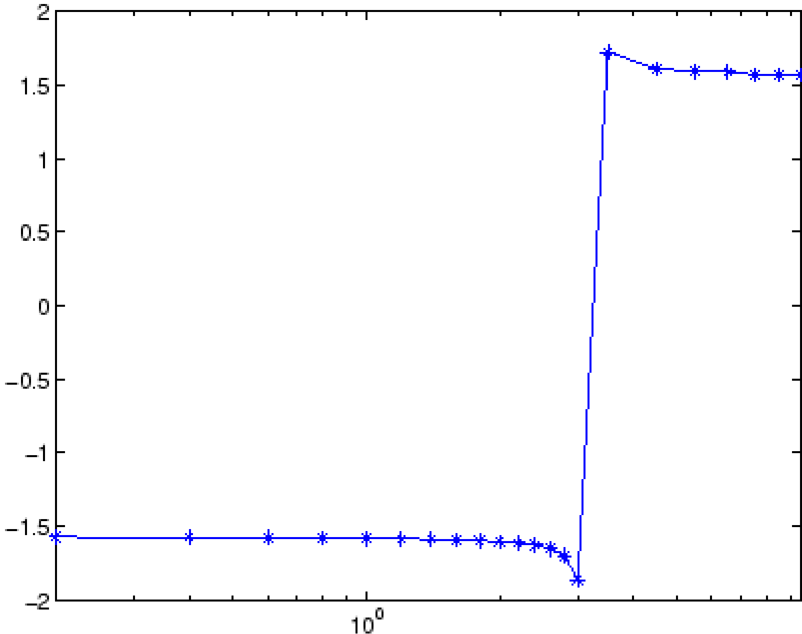
Examples

Example 1

The following phase data comes from the frequency response of a third-order transfer function. The phase curve jumps `3.5873` radians between `w = 3.0` and `w = 3.5`, from `-1.8621` to `1.7252`.

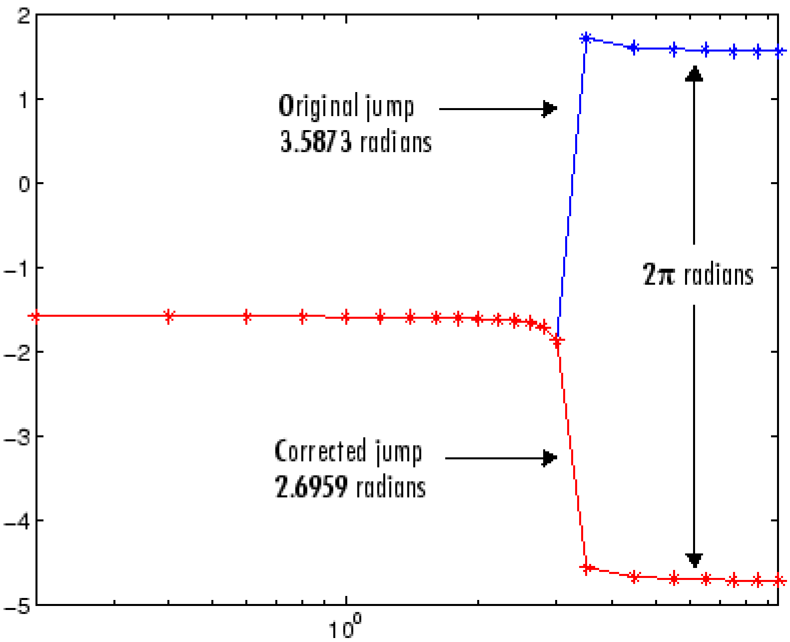
```
w = [0:.2:3,3.5:1:10];
p = [    0
     -1.5728
     -1.5747
     -1.5772
```

```
      -1.5790
      -1.5816
      -1.5852
      -1.5877
      -1.5922
      -1.5976
      -1.6044
      -1.6129
      -1.6269
      -1.6512
      -1.6998
      -1.8621
       1.7252
       1.6124
       1.5930
       1.5916
       1.5708
       1.5708
       1.5708 ];
semilogx(w,p,'b*-'), hold
```



Using `unwrap` to correct the phase angle, the resulting jump is `2.6959`, which is less than the default jump tolerance π. This figure plots the new curve over the original curve.

```
semilogx(w,unwrap(p),'r*-')
```



Example 2

Array `P` features smoothly increasing phase angles except for discontinuities at elements `(3,1)` and `(1,2)`.

```
P = [       0     7.0686     1.5708     2.3562
       0.1963     0.9817     1.7671     2.5525
       6.6759     1.1781     1.9635     2.7489
       0.5890     1.3744     2.1598     2.9452 ]
```

The function `Q = unwrap(P)` eliminates these discontinuities.

```
Q =
            0     7.0686     1.5708     2.3562
       0.1963     7.2649     1.7671     2.5525
       0.3927     7.4613     1.9635     2.7489
       0.5890     7.6576     2.1598     2.9452
```

See Also

`abs` | `angle`

unzip

Purpose

Extract contents of zip file

Syntax

```
unzip(zipfilename)
unzip(zipfilename,outputdir)
filenames = unzip(zipfilename,outputdir)
```

Description

`unzip(zipfilename)` extracts the archived contents of `zipfilename` into the current folder, preserving the files' attributes and timestamps. The `unzip` function can extract files from your local system or files from an Internet URL.

`unzip(zipfilename,outputdir)` extracts the contents of `zipfilename` into the folder `outputdir`.

`filenames = unzip(zipfilename,outputdir)` returns the names of the extracted files in the string cell array `filenames`. Specifying `outputdir` is optional.

Tips

- `unzip` does not support password-protected or encrypted zip archives.
- If any files in the target folder have the same name as files in the zip file, and you have write permission to the files, `unzip` overwrites the existing files with the archived versions. If you do not have write permission, `unzip` issues a warning.
- Extract files that contain non-7-bit ASCII characters on a machine that has the appropriate language/encoding settings.

Input Arguments

zipfilename

String that specifies the name of the zip file.

If `zipfilename` does not include the full path, `unzip` searches for the file in the current folder and along the MATLAB path. If you do not specify the file extension, `unzip` appends `.zip`.

If you are downloading a URL, `zipfilename` must include the protocol type (for example, `http://`). The `unzip` function downloads the URL to the temporary folder on your system, and deletes the URL on cleanup.

outputdir

String that specifies the target folder for the extracted files.

Default: current folder ('.')

Output Arguments

entrynames

Cell array of strings that contain the paths of the extracted files.

If `outputdir` specifies a relative path, `filenames` contains the relative path. If `outputdir` specifies an absolute path, `filenames` contains the absolute path.

Examples

Copy the example MAT-files to the folder `archive`:

```
% Zip the example MAT-files to examples.zip
zip('examples.zip','*.mat',...
    fullfile(matlabroot,'toolbox','matlab','demos'))

% Unzip examples.zip to the folder 'archive'
unzip('examples','archive')
```

Download Cleve Moler's "Numerical Computing with MATLAB" examples to the output folder `ncm`:

```
url ='http://www.mathworks.com/moler/ncm.zip';
ncmFiles = unzip(url,'ncm')
```

Alternatives

To extract files from a zip file using the Current Folder browser, select the zip file, right-click to open the context menu, and then select **Extract**.

See Also

`fileattrib` | `gzip` | `gunzip` | `tar` | `untar` | `zip`

uplus

Purpose

Unary plus

Syntax

```
b = +a
b = uplus(a)
```

Description

`b = +a` for numeric arrays is `a`.

`b = uplus(a)` is called for the syntax `+a` when `a` is an object.

See Also

`uminus`

Purpose

Convert string to uppercase

Syntax

```
t  = upper('str')
B = upper(A)
```

Description

`t = upper('str')` converts any lowercase characters in the string `str` to the corresponding uppercase characters and leaves all other characters unchanged.

`B = upper(A)` when `A` is a cell array of strings, returns a cell array the same size as `A` containing the result of applying `upper` to each string within `A`.

Examples

`upper('attention!')` is `ATTENTION!`.

Tips

Character sets supported:

- PC: Windows Latin-1
- Other: ISO Latin-1 (ISO 8859-1)

See Also

`lower`

Purpose

Download URL content to MATLAB string

Syntax

```
str = urlread(URL)
str = urlread(URL,Name,Value)
[str,status] = urlread( ___ )
```

Description

`str = urlread(URL)` downloads the HTML Web content from the specified `URL` into the string `str`. `urlread` does not retrieve hyperlink targets and images.

`str = urlread(URL,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[str,status] = urlread( ___ )` suppresses the display of error messages, using any of the input arguments in the previous syntaxes. When the operation is successful, `status` is `1`. Otherwise, `status` is `0`

Input Arguments

URL - Content location

string

Content location, specified as a string. Include the transfer protocol, such as `http`, `ftp`, or `file`.

Example: `'http://www.mathworks.com/matlabcentral'`

Name-Value Pair Arguments

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

Example: `'Timeout',10,'Charset','UTF-8'` specifies that `urlread` should time out after 10 seconds, and the character encoding of the file is UTF-8.

'Get' - Data to send to the Web form using the GET method

cell array

Parameters of the data to send to the Web form using the GET method, specified as the comma-separated pair consisting of `'get'` and a cell array of paired parameter names and values. The supported parameters depend upon the URL.

`'Get'` includes the data in the URL, separated by `?` and `&` characters.

Example: `'Get',{'term','urlread'}`

'Post' - Data to send to the Web form using the POST method

cell array

Parameters of the data to send to the Web form using the POST method, specified as the comma-separated pair consisting of `'post'` and a cell array of paired parameter names and values. The supported parameters depend upon the URL.

`'Post'` submits the data as part of the request headers, not explicitly in the URL.

'Charset' - Character encoding

string

Character encoding, specified as the comma-separated pair consisting of `'Charset'` and a string. If you do not specify `Charset`, the function attempts to determine the character encoding from the headers of the file. If the character encoding cannot be determined, `Charset` defaults to the native encoding for the file protocol, and UTF-8 for all other protocols.

Example: `'Charset','ISO-8859-1'`

'Timeout' - Timeout duration

scalar

Timeout duration in seconds, specified as the comma-separated pair consisting of `'Timeout'` and a scalar. The timeout duration determines when the function errors rather than continues to wait for the server to respond or send data.

Example: `'Timeout',10`

'UserAgent' - Client user agent identification

string

Client user agent identification, specified as the comma-separated pair consisting of `'UserAgent'` and a string.

Example: `'UserAgent','MATLAB R2012b'`

'Authentication' - HTTP authentication mechanism

`'Basic'`

HTTP authentication mechanism, specified as the comma-separated pair consisting of `'Authentication'` and a string. Currently, only the value `'Basic'` is supported. `'Authentication','Basic'` specifies basic authentication.

If you include the `Authentication` argument, you must also include the `Username` and `Password` arguments.

'Username' - User identifier

string

User identifier, specified as the comma-separated pair consisting of `'Username'` and a string. If you include the `Username` argument, you must also include the `Password` and `Authentication` arguments.

Example: `'Username','myName'`

'Password' - User authentication password

string

User authentication password, specified as the comma-separated pair consisting of `'Password'` and a string. If you include the `Password` argument, you must also include the `Username` and `Authentication` arguments.

Example: `'Password','myPassword123'`

Output Arguments

str - Contents of the file at the specified URL

string

Contents of the file at the specified URL, returned as a string. For example, if the URL corresponds to an HTML page, `str` contains the text and markup in the HTML file. If the URL corresponds to a binary file, `str` is not readable.

status - Download status

1 | 0

Download status, returned as either `1` or `0`. When the download is successful, `status` is `1`. Otherwise, `status` is `0`.

Examples

Download Web Content by Specifying Complete URL

Download the HTML for the page on the MATLAB Central File Exchange that lists submissions related to `urlread`.

```
fullURL = ['http://www.mathworks.com/matlabcentral/fileexchange' ...
    '?term=urlread'];
str = urlread(fullURL);
```

`urlread` reads from the specified URL and downloads the HTML content to the string `str`.

Download Web Content Related to a Term

Download the HTML for the page on the MATLAB Central File Exchange that lists submissions related to `urlread`.

```
URL = 'http://www.mathworks.com/matlabcentral/fileexchange';
str = urlread(URL,'Get',{'term','urlread'});
```

`urlread` reads from http://www.mathworks.com/matlabcentral/fileexchange/?term=urlread and downloads the HTML content to the string `str`.

Specify Timeout Duration

Download content from a page on the MATLAB Central File Exchange as in the first example, and specify a timeout duration of 5 seconds.

```
fullURL = ['http://www.mathworks.com/matlabcentral/fileexchange' ...
    '?term=urlread'];
str = urlread(fullURL,'Timeout',5);
```

Tips

- `urlread` saves Web content to a string. To save content to a file, use `urlwrite`.
- `urlread` and `urlwrite` can download content from FTP sites. Alternatively, use the `ftp` function to connect to an FTP server and the `mget` function to download a file.

See Also

`urlwriteftp` **|** | `webmget` **|**

Concepts

- "Specify Proxy Server Settings for Connecting to the Internet"

urlwrite

Purpose

Download URL content and save to file

Syntax

```
urlwrite(URL,filename)
urlwrite(URL,filename,Name,Value)

[filestr,status] = urlwrite( ___ )
```

Description

`urlwrite(URL,filename)` reads Web content at the specified `URL` and saves it to the file specified by `filename`.

`urlwrite(URL,filename,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[filestr,status] = urlwrite( ___ )` stores the file path in variable `filestr`, and suppresses the display of error messages, using any of the input arguments in the previous syntaxes. When the operation is successful, `status` is `1`. Otherwise, `status` is `0`.

Input Arguments

URL - Content location

string

Content location, specified as a string. Include the transfer protocol, such as `http`, `ftp`, or `file`.

Example: `'http://www.mathworks.com/matlabcentral'`

filename - Name of file to store Web content

string

Name of the file to store the Web content, specified as a string. If you do not specify the path for `filename`, `urlwrite` saves the file in the current folder.

Example: `'myfile.html'`

Name-Value Pair Arguments

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

Example: `'Timeout',10,'Charset','UTF-8'` specifies that `urlread` should time out after 10 seconds, and the character encoding of the file is UTF-8.

'Get' - Data to send to the Web form using the GET method

cell array

Parameters of the data to send to the Web form using the GET method, specified as the comma-separated pair consisting of `'get'` and a cell array of paired parameter names and values. The supported parameters depend upon the URL.

`'Get'` includes the data in the URL, separated by `?` and `&` characters.

Example: `'Get',{'term','urlread'}`

'Post' - Data to send to the Web form using the POST method

cell array

Parameters of the data to send to the Web form using the POST method, specified as the comma-separated pair consisting of `'post'` and a cell array of paired parameter names and values. The supported parameters depend upon the URL.

`'Post'` submits the data as part of the request headers, not explicitly in the URL.

'Charset' - Character encoding

string

Character encoding, specified as the comma-separated pair consisting of `'Charset'` and a string. If you do not specify `Charset`, the function attempts to determine the character encoding from the headers of the

file. If the character encoding cannot be determined, `Charset` defaults to the native encoding for the file protocol, and UTF-8 for all other protocols.

Example: `'Charset','ISO-8859-1'`

'Timeout' - Timeout duration

scalar

Timeout duration in seconds, specified as the comma-separated pair consisting of `'Timeout'` and a scalar. The timeout duration determines when the function errors rather than continues to wait for the server to respond or send data.

Example: `'Timeout',10`

'UserAgent' - Client user agent identification

string

Client user agent identification, specified as the comma-separated pair consisting of `'UserAgent'` and a string.

Example: `'UserAgent','MATLAB R2012b'`

'Authentication' - HTTP authentication mechanism

`'Basic'`

HTTP authentication mechanism, specified as the comma-separated pair consisting of `'Authentication'` and a string. Currently, only the value `'Basic'` is supported. `'Authentication','Basic'` specifies basic authentication.

If you include the `Authentication` argument, you must also include the `Username` and `Password` arguments.

'Username' - User identifier

string

User identifier, specified as the comma-separated pair consisting of `'Username'` and a string. If you include the `Username` argument, you must also include the `Password` and `Authentication` arguments.

Example: `'Username','myName'`

'Password' - User authentication password

string

User authentication password, specified as the comma-separated pair consisting of `'Password'` and a string. If you include the `Password` argument, you must also include the `Username` and `Authentication` arguments.

Example: `'Password','myPassword123'`

Output Arguments

filestr - Path of the file

string

Path of the file specified by `filename`, returned as a string.

status - Download status

1 | 0

Download status, returned as either `1` or `0`. When the download is successful, `status` is `1`. Otherwise, `status` is `0`.

Examples

Download Web Content by Specifying Complete URL

Download the HTML for the page on the MATLAB Central File Exchange that lists submissions related to `urlwrite`. Save the results to `samples.html` in the current directory.

```
fullURL = ['http://www.mathworks.com/matlabcentral/fileexchange' ...
           '?term=urlwrite'];
filename = 'samples.html';
urlwrite(fullURL,filename);
```

View the file.

```
web(filename)
```

Download Web Content Related to a Term

Download the HTML for the page on the MATLAB Central File Exchange that lists submissions related to `urlwrite`. Save the results to `samples.html` in the current directory.

```
URL = 'http://www.mathworks.com/matlabcentral/fileexchange';
filename = 'samples.html';
urlwrite(URL,filename,'get',{'term','urlwrite'});
```

`urlwrite` downloads the HTML content from http://www.mathworks.com/matlabcentral/fileexchange/?term=urlwrite and writes it to `samples.html`.

Specify Timeout Duration

Download content from a page on the MATLAB Central File Exchange as in the first example, and specify a timeout duration of 5 seconds.

```
fullURL = ['http://www.mathworks.com/matlabcentral/fileexchange' ...
    '?term=urlwrite'];
filename = 'samples.html';
urlwrite(fullURL,filename,'Timeout',5);
```

Tips

- `urlread` and `urlwrite` can download content from FTP sites. Alternatively, use the `ftp` function to connect to an FTP server and the `mget` function to download a file.

See Also

`urlreadmget` **|** | `webftp` **|**

Concepts

- “Specify Proxy Server Settings for Connecting to the Internet”

Purpose

Determine if Java feature is supported in MATLAB

Syntax

```
tf = usejava(feature)
```

Description

`tf = usejava(feature)` returns logical `1` (`true`) if the specified feature is supported. Otherwise, it returns logical `0` (`false`).

Input Arguments

feature - Java feature

`'awt'` | `'desktop'` | `'jvm'` | `'swing'`

Java feature, specified as one of these values:

`'awt'`	Java GUI components in the Abstract Window Toolkit (AWT) components are available.
`'desktop'`	The MATLAB interactive desktop is running.
`'jvm'`	The Java Virtual Machine software (JVM) is running.
`'swing'`	Swing components (Java lightweight GUI components in the Java Foundation Classes) are available.

Examples

Display Error Message

Use the following code snippet to test that the AWT GUI components are available before attempting to display a Java Frame.

```
if usejava('awt')
   myFrame = java.awt.Frame;
else
   disp('Unable to open a Java Frame');
end
```

If the AWT is not available on your system, MATLAB displays the message.

Call error Function

Use the following code snippet to terminate a script if MATLAB does not have access to JVM software.

The variable, `filename`, is a function that contains Java code.

```
if ~usejava('jvm')
   error([filename ' requires Java to run.']);
end
```

See Also

`javachk` | `error`

userpath

Purpose

View or change user portion of search path

Syntax

```
userpath

userpath(newpath)
userpath('reset')
userpath('clear')
```

Description

`userpath` returns a string specifying the first folder on the search path. MATLAB adds the *`userpath`* to the search path upon startup.

`userpath(newpath)` sets the primary *`userpath`* folder to `newpath`. The `newpath` folder appears at the top of the search path immediately and at startup in future sessions. MATLAB removes the folder previously specified by *`userpath`* from the search path.

`userpath('reset')` sets the primary *`userpath`* folder to the default for that platform, creating the `Documents/MATLAB` (or `My Documents/MATLAB`) folder, if it does not exist. MATLAB immediately adds the default folder to the top of the search path, and also adds it to the search path at startup in future sessions. On Linux, the `MATLAB` directory is not created if the `Documents` directory does not exist.

`userpath('clear')` clears the value for the primary *`userpath`* immediately, and for future MATLAB sessions. MATLAB removes the folder previously specified by *`userpath`* from the search path.

Input Arguments

newpath - New userpath value
string

New userpath value, specified as a string. `newpath` must be an absolute path.

Example: `'C:\myFolder'`

Examples

View userpath Folder

This example assumes `userpath` is set to the default value on the Windows XP platform, `My Documents\MATLAB`. Start MATLAB and display the current folder:

```
pwd

H:\My Documents\MATLAB
```

In this example, `H` is the drive at which `My Documents` is located.

Confirm that the current folder is the *`userpath`*.

```
userpath

H:\My Documents\MATLAB;
```

Display the search path.

```
path

MATLABPATH

 H:\My Documents\MATLAB
 C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
 C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...
```

MATLAB returns the search path. The *`userpath`* portion is at the top.

Set New Value for userpath

Assume *`userpath`* is set to the default value on the Windows XP platform, `My Documents\MATLAB`. Change the value from the default for *`userpath`* to `C:\Research_Project`.

```
newpath = 'C:\Research_Project';
userpath(newpath)
```

View the effect of the change on the search path.

```
path
```

```
MATLABPATH

 C:\Research_Project
 C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
 C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...
```

MATLAB displays the search path, with the new value for *userpath* portion at the top. Note that MATLAB automatically removed the previous value of *userpath*, `H:\My Documents\MATLAB`, from the search path when you assigned a new value to *userpath*.

Clear the Value for userpath

Assume `userpath` is set to the default value and you do not want any folders to be added to the search path upon startup. Confirm the default is currently set.

```
userpath
```

```
H:\My Documents\MATLAB;
```

Verify that the `userpath` folder is at the top of the search path.

```
path
```

```
MATLABPATH

H:\My Documents\MATLAB
C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...
```

Clear the value.

```
userpath('clear')
```

Verify the result.

```
userpath
```

```
ans =
     ''
```

Confirm the *userpath* folder was removed from the search path.

```
path
```

```
MATLABPATH

C:\Program Files\MATLAB\R2009a\toolbox\matlab\general
C:\Program Files\MATLAB\R2009a\toolbox\matlab\ops
...
```

Note By default, the startup folder is the *userpath* folder when you start MATLAB by double-clicking the shortcut on Windows platforms, or by double-clicking the application on Macintosh platforms. If you clear the value for *userpath*, the startup folder will not necessarily be on the search path. Removing the *userpath* folder from the search path *and* saving the changes to the path has the same effect.

Tips

- On Macintosh and UNIX platforms, you can automatically add subfolders to the top of the search path upon startup by specifying the path for the subfolders via the `MATLABPATH` environment variable.

See Also

`addpath` | `path` | `rmpath` | `savepath` | `startup`

Concepts

- “What Is the MATLAB Search Path?”
- “MATLAB Startup Folder”

validateattributes

Purpose

Check validity of array

Syntax

```
validateattributes(A,classes,attributes)
validateattributes(A,classes,attributes,argIndex)
validateattributes(A,classes,attributes,funcName)
validateattributes(A,classes,attributes,funcName,varName)
validateattributes(A,classes,attributes,funcName,varName,
   argIndex)
```

Description

`validateattributes(A,classes,attributes)` validates that array `A` belongs to at least one of the specified classes (or its subclass) and has all of the specified attributes. If `A` does not meet the criteria, MATLAB throws an error and displays a formatted error message. Otherwise, `validateattributes` completes without displaying any output.

`validateattributes(A,classes,attributes,argIndex)` includes the position of the input in your function argument list as part of any generated error messages.

`validateattributes(A,classes,attributes,funcName)` includes the specified function name in generated error identifiers.

`validateattributes(A,classes,attributes,funcName,varName)` includes the specified variable name in generated error messages.

`validateattributes(A,classes,attributes,funcName,varName,argIndex)` includes the specified information in generated error messages or identifiers.

Input Arguments

A - Input

any type of array

Input, specified as any type of array.

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

classes - Valid data types

cell array of strings

Valid data types, specified as a cell array of strings. Each string can be the name of any built-in or custom class, including:

`'single'`	Single-precision number
`'double'`	Double-precision number
`'int8'`	Signed 8-bit integer
`'int16'`	Signed 16-bit integer
`'int32'`	Signed 32-bit integer
`'int64'`	Signed 64-bit integer
`'uint8'`	Unsigned 8-bit integer
`'uint16'`	Unsigned 16-bit integer
`'uint32'`	Unsigned 32-bit integer
`'uint64'`	Unsigned 64-bit integer
`'logical'`	Logical `true` or `false`
`'char'`	Character or string
`'struct'`	Structure array
`'cell'`	Cell array
`'function_handle'`	function handle

'numeric'	Any value for which the isnumeric function returns true, including int8, int16, int32, int64, uint8, uint16, uint32, uint64, single, or double
'<*class_name*>'	Any other class name

Data Types
cell

attributes - Valid attributes

cell array of strings

Valid attributes, specified as a cell array of strings.

Some attributes also require numeric values, such as attributes that specify the size or number of elements of A. For these attributes, the numeric value or vector must immediately follow the attribute name string in the cell array.

Attributes that describe the size and shape of array A:

'2d'	Two-dimensional array, including scalars, vectors, matrices, and empty arrays
'3d'	Array with three or fewer dimensions
'column'	Column vector, N-by-1
'row'	Row vector, 1-by-N
'scalar'	Scalar value, 1-by-1
'vector'	Row or column vector, or a scalar value
'size', [d1,...,dN]	Array with dimensions d1-by-...-by-dN. To skip checking a particular dimension, specify NaN for that dimension, such as [3,4,NaN,2].
'numel', N	Array with N elements
'ncols', N	Array with N columns

'nrows', N	Array with N rows
'ndims', N	N-dimensional array
'square'	Square matrix; in other words, a two-dimensional array with equal number of rows and columns
'diag'	Diagonal matrix
'nonempty'	No dimensions that equal zero
'nonsparse'	Array that is not sparse

Attributes that specify valid ranges for values in A:

'>', N	All values greater than N
'>=', N	All values greater than or equal to N
'<', N	All values less than N
'<=', N	All values less than or equal to N

Attributes that check types of values in a numeric or logical array, A:

'binary'	Array of ones and zeros
'even'	Array of even integers (includes zero)
'odd'	Array of odd integers
'integer'	Array of integer values
'real'	Array of real values
'finite'	Array of finite values
'nonnan'	No NaN (Not a Number) elements
'nonnegative'	No elements less than zero

'nonzero'	No elements equal to zero
'positive'	No elements less than or equal to zero

Data Types
`cell`

funcName - Name of function whose input you are validating

string

Name of function whose input you are validating, specified as a string. If you specify an empty string, `''`, the `validateattribute` function ignores the `funcName` input.

Data Types
`char`

varName - Name of input variable

string

Name of input variable, specified as a string. If you specify an empty string, `''`, the `validateattribute` function ignores the `varName` input.

Data Types
`char`

argIndex - Position of input argument

positive integer

Position of input argument, specified as a positive integer.

Data Types
`double`

Examples

Validate the Size of an Array

```
classes = {'numeric'};
attributes = {'size',[4,6,2]};

A = rand(3,5,2);
```

```
validateattributes(A,classes,attributes)
```

```
Expected input to be of size 4x6x2 when it is actually size 3x5x2.
```

Because A did not match the specified attributes, MATLAB throws an error message.

Check the Attributes of a Complex Number

Assuming that a is to be the second input argument to a function, check that it is nonnegative.

```
a = complex(1,1);
validateattributes(a,{'numeric'},{'nonnegative'},2);
```

```
Expected input number 2 to be nonnegative.
```

Because complex numbers lack a well-defined ordering in the complex plane, validateattributes does not recognize them as positive or negative.

Ensure Array Values Are Within a Specified Range

Check that the values in an array are 8-bit integers between 0 and 10.

Assume that this code occurs in a function called Rankings.

```
classes = {'uint8','int8'};
attributes = {'>',0,'<',10};
funcName = 'Rankings';
A = int8(magic(4));

validateattributes(A,classes,attributes,funcName)
```

```
Error using Rankings
Expected input to be an array with all of the values < 10.
```

Validate Function Input Parameters Using `inputParser`

Create a custom function that checks input parameters with `inputParser`, and use `validateattributes` as the validating function for the `addRequired` and `addOptional` methods.

Define the function.

```
function a = findArea(shape,dim1,varargin)
   charchk = {'char'};
   numchk = {'double'};
   nempty = {'nonempty'};

   validateattributes(shape,charchk,nempty,'findArea','shape');
   validateattributes(dim1,numchk,nempty,'findArea','dim1');
   validateattributes(varargin,numchk,nempty,'findArea','varargin');

   switch shape
      case 'circle'
         a = pi * dim1.^2;
      case 'rectangle'
         a = dim1 .* p.Results.dim2;
   end
```

Call the function with a nonnumeric third input.

```
myarea = findArea('rectangle',3,'x')
```

```
Error using findArea
Expected varargin to be one of these types:

double

Instead its type was cell.

Error in findArea (line 8)
   validateattributes(varargin,numchk,nempty,'findArea','varargin');
```

Validate Arguments of a Function

Check the inputs of a function and include information about the input name and position in generated error.

Define the function.

```
function v = findVolume(shape,ht,wd,ln)
   validateattributes(shape,{'char'},{'nonempty'},mfilename,'Shape',1)
   validateattributes(ht,{'numeric'},{'nonempty'},mfilename,'Height',2
   validateattributes(wd,{'numeric'},{'nonempty'},mfilename,'Width',3)
   validateattributes(ln,{'numeric'},{'nonempty'},mfilename,'Length',4
```

Call the function without the shape input string.

```
vol = findVolume(10,7,4)
```

```
Error using findVolume
Expected input number 1, Shape, to be one of these types:

char

Instead its type was double.

Error in findVolume (line 2)
validateattributes(shape,{'char'},{'nonempty'},mfilename,'Shape',1)
```

The function name becomes part of the error identifier.

```
MException.last.identifier
```

```
ans =

MATLAB:findVolume:invalidType
```

See Also

validatestring | is* | isa | isnumericinputParser |

validatestring

Purpose

Check validity of text string

Syntax

```
validStr = validatestring(str,validStrings)
validStr = validatestring(str,validStrings,argIndex)
validStr = validatestring(str,validStrings,funcName)
validStr =
validatestring(str,validStrings,funcName,varName)
validStr =
validatestring(str,validStrings,funcName,varName,
   argIndex)
```

Description

`validStr = validatestring(str,validStrings)` checks the validity of text string `str`. If `str` is an unambiguous, case-insensitive match to a string in cell array `validStrings`, the `validatestring` function returns the matching string in `validstr`. Otherwise, MATLAB throws an error and issues a formatted error message.

`validStr = validatestring(str,validStrings,argIndex)` includes the position of the input in your function argument list as part of any generated error messages.

`validStr = validatestring(str,validStrings,funcName)` includes the specified function name in generated error identifiers.

`validStr =`
`validatestring(str,validStrings,funcName,varName)` includes the specified variable name in generated error messages.

`validStr =`
`validatestring(str,validStrings,funcName,varName,argIndex)` includes the specified information in generated error messages or identifiers.

Input Arguments

str

String to validate, of type `char`.

validStrings

Cell array of allowed strings.

funcName

String that specifies the name of the function whose input you are validating. If you specify an empty string, `''`, the `validatestring` function ignores the `funcname` input.

varName

String that specifies the name of the input variable. If you specify an empty string, `''`, the `validatstring` function ignores the `varName` input.

argIndex

Positive integer that specifies the position of the input argument.

Output Arguments

validStr

String that contains the element of `validStrings` that is an unambiguous, case-insensitive match to `str`.

Example — Match 'ball' with . . .	Return Value	Type of Match
`ball`, `barn`, `bell`	`ball`	Exact match
`balloon`, `barn`	`balloon`	Partial match (leading characters)
`ballo`, `balloo`, `balloon`	`ballo` (shortest match)	Multiple partial matches where each string is a subset of another
`balloon`, `ballet`	Error	Multiple partial matches to unique strings
`barn`, `bell`	Error	No match

Examples

Check whether a string is in a set of valid values.

```
str = 'won';
validStrings = {'wind','wonder','when'};

validStr = validatestring(str,validStrings)
```

Because `str` is a partial match to a unique string, this code returns

```
validStr =
     wonder
```

However, if `str` is a partial match to multiple strings, and the strings are not subsets of each other, `validatestring` throws an error and displays a formatted message.

```
str = 'won';
validStrings = {'wonder','wondrous','wonderful'};

validStr = validatestring(str,validStrings)
```

The error message is

```
Expected argument to match one of these strings:

  wonder, wondrous, wonderful

The input, 'won', matched more than one valid string.
```

Check inputs to a custom function, and include information about the input name and position in generated errors.

```
function a = findArea(shape,ht,wd,units)
   a = 0;
   expectedShapes = {'square','rectangle','triangle'};
   expectedUnits = {'cm','m','in','ft','yds'};
```

```
shapeName = validatestring(shape,expectedShapes,mfilename,'Shape',1
unitAbbrev = validatestring(units,expectedUnits,mfilename,'Units',4
```

When you call the function with valid input strings,

```
myarea = findArea('rect',10,3,'cm')
```

the function stores the inputs in local variables `shapeName` and `unitAbbrev`:

```
shapeName =
     rectangle

unitAbbrev =
     cm
```

However, if the inputs are not valid, such as

```
myarea = findArea('circle',10,3,'cm')
```

MATLAB displays

```
Error using findArea
Expected argument 1, Shape, to match one of these strings:

  square, rectangle, triangle

The input, 'circle', did not match any of the valid strings.
```

The function name is part of the error identifier, so

```
MException.last.identifier
```

returns

```
MATLAB:findArea:unrecognizedStringChoice
```

See Also

`validateattributes` | `is*` | `isa` | `inputParser`

containers.Map.values

Purpose

Identify values in `containers.Map` object

Syntax

```
valueSet = values(mapObj)
valueSet = values(mapObj,keySet)
```

Description

`valueSet = values(mapObj)` returns all of the values in `mapObj`.

`valueSet = values(mapObj,keySet)` returns values that correspond to the specified keys.

Input Arguments

mapObj

Object of class `containers.Map`.

keySet

Cell array that specifies keys in `mapObj`.

Output Arguments

valueSet

Cell array containing values from `mapObj`. If you specify `keySet`, the `valueSet` array has the same size and dimensions as `keySet`.

Examples

Get All Values in a Map

Create a map, and view all values in the map:

```
myKeys = {'a','b','c'};
myValues = [1,2,3];
mapObj = containers.Map(myKeys,myValues);

valueSet = values(mapObj)
```

This code returns 1-by-3 cell array `valueSet`:

```
valueSet =
    [1]    [2]    [3]
```

Get Selected Values in a Map

View the values corresponding to keys a and c in mapObj, created in the previous example:

```
keySet = {'a','c'};
valueSet = values(mapObj,keySet)
```

This code returns 1-by-2 cell array valueSet:

```
valueSet =
    [1]    [3]
```

See Also

containers.Map | keys | isKey

Purpose

Vandermonde matrix

Syntax

```
A = vander(v)
```

Description

`A = vander(v)` returns the Vandermonde matrix whose columns are powers of the vector `v`, that is, `A(i,j) = v(i)^(n-j)`, where `n = length(v)`.

Examples

```
vander(1:.5:3)

ans =

    1.0000    1.0000    1.0000    1.0000    1.0000
    5.0625    3.3750    2.2500    1.5000    1.0000
   16.0000    8.0000    4.0000    2.0000    1.0000
   39.0625   15.6250    6.2500    2.5000    1.0000
   81.0000   27.0000    9.0000    3.0000    1.0000
```

See Also

`gallery`

Purpose

Variance

Syntax

```
V = var(X)
V = var(X,1)
V = var(X,w)
V = var(X,w,dim)
```

Description

`V = var(X)` returns the variance of `X` for vectors. For matrices, `var(X)`is a row vector containing the variance of each column of `X`. For `N`-dimensional arrays, `var` operates along the first nonsingleton dimension of `X`. The result `V` is an unbiased estimator of the variance of the population from which `X` is drawn, as long as `X` consists of independent, identically distributed samples.

`var` normalizes `V` by `N 1` if `N > 1`, where `N` is the sample size. This is an unbiased estimator of the variance of the population from which `X` is drawn, as long as `X` consists of independent, identically distributed samples. For `N = 1`, `V` is normalized by 1.

`V = var(X,1)` normalizes by `N` and produces the second moment of the sample about its mean. `var(X,0)` is equivalent to `var(X)`.

`V = var(X,w)` computes the variance using the weight vector `w`. The length of `w` must equal the length of the dimension over which `var` operates, and its elements must be nonnegative. If `X(i)` is assumed to have variance proportional to 1/`w(i)`, then `V * mean(w)/w(i)` is an estimate of the variance of `X(i)`. In other words, `V * mean(w)` is an estimate of variance for an observation given weight 1.

`V = var(X,w,dim)` takes the variance along the dimension `dim` of `X`. Pass in `0` for `w` to use the default normalization by N – 1, or 1 to use `N`.

The variance is the square of the standard deviation (STD).

Examples

Create a matrix and find the variance along the dimensions.

```
X = [4 -2 1; 9 5 7]
var(X,0,1)
ans =
```

```
   12.5000   24.5000   18.0000

var(X,0,2)
ans =
     9
     4
```

See Also

`corrcoef` | `cov` | `mean` | `median` | `std`

Purpose

Variable-length input argument list

Syntax

```
varargin
```

Description

`varargin` is an input variable in a function definition statement that allows the function to accept any number of input arguments. Specify `varargin` using lowercase characters, and include it as the last input argument after any explicitly declared inputs. When the function executes, `varargin` is a 1-by-N cell array, where N is the number of inputs that the function receives after the explicitly declared inputs.

Examples

Variable Number of Function Inputs

Define a function in a file named `varlist.m` that accepts a variable number of inputs and displays the values of each input.

```
function varlist(varargin)
   fprintf('Number of arguments: %d\n',nargin);
   celldisp(varargin)
```

Call `varlist` with several inputs.

```
varlist(ones(3),'some text',pi)
```

```
Number of arguments: 3

varargin{1} =
     1     1     1
     1     1     1
     1     1     1

varargin{2} =
some text

varargin{3} =
    3.1416
```

varargin and Declared Inputs

Define a function in a file named `varlist2.m` that expects inputs `X` and `Y`, and accepts a variable number of additional inputs.

```
function varlist2(X,Y,varargin)
   fprintf('Total number of inputs = %d\n',nargin);

   nVarargs = length(varargin);
   fprintf('Inputs in varargin(%d):\n',nVarargs)
   for k = 1:nVarargs
      fprintf('   %d\n', varargin{k})
   end
```

Call `varlist2` with more than two inputs.

```
varlist2(10,20,30,40,50)
```

```
Total number of inputs = 5
Inputs in varargin(3):
   30
   40
   50
```

See Also

`varargout` | `nargin` | `nargout` | `narginchk` | `nargoutchk` | `inputname`

Related Examples

- "Support Variable Number of Inputs"

Concepts

- "Argument Checking in Nested Functions"

Purpose

Variable-length output argument list

Syntax

```
varargout
```

Description

`varargout` is an output variable in a function definition statement that allows the function to return any number of output arguments. Specify `varargout` using lowercase characters, and include it as the last output argument after any explicitly declared outputs. When the function executes, `varargout` is a 1-by-*N* cell array, where *N* is the number of outputs requested after the explicitly declared outputs.

Examples

Variable Number of Function Outputs

Define a function in a file named `sizeout.m` that returns an output size vector `s` and a variable number of additional scalar values.

```
function [s,varargout] = sizeout(x)
nout = max(nargout,1) - 1;
s = size(x);
for k=1:nout
   varargout{k} = s(k);
end
```

Output `s` contains the dimensions of the input array `x`. Additional outputs correspond to the individual dimensions within `s`.

Call `sizeout` on a three-dimensional array and request three outputs.

```
[s,rows,cols] = sizeout(rand(4,5,2))
```

```
s =
     4     5     2

rows =
     4

cols =
     5
```

See Also

varargin | nargout | nargin | nargoutchk | narginchk | inputname

Purpose

Vectorize expression

Syntax

```
vectorize(s)
vectorize(f)
```

Description

`vectorize(s)` where `s` is a string expression, inserts a `.` before any `^`, `*` or `/` in `s`. The result is a character string.

Note `vectorize` will not accept inline function objects (`f`) in a future release.

`vectorize(f)` where `f` is an inline function object, vectorizes the formula for `f`. The result is the vectorized version of the function.

See Also

`inline` | `cd` | `dbtype` | `delete` | `dir` | `path` | `what` | `who`

Purpose

Version information for MathWorks products

Syntax

```
ver
ver product

product_info = ver(product)
```

Description

`ver` displays:

- A header containing the current MATLAB product family version number, license number, operating system, and version of Java software for the MATLAB product.
- The version numbers for MATLAB and all other installed MathWorks products.

`ver product` displays, in addition to the header information:

- The current version number for `product`, where `product` is the name of the folder that contains the `Contents.m` file for the product you are inquiring about.

`product_info = ver(product)` returns product information to the structure array, `product_info`.

Input Arguments

product - product-specific information

The product or toolbox for which you want to view version information, specified as a string.

To determine the string to use, run the following code, substituting the name of a product function for *toolbxfcn*:

```
n = 'toolbxfcn';
pat = '(?<=^.+[\\/]toolbox[\\/])[^\\/]+';
regexp(which(n), pat, 'match', 'once')
```

Output Arguments

product_info - product name, version, release, and date

Product name, version, release, and date, returned as a structure array with these fields: `Name`, `Version`, `Release`, and `Date`. If a license is a trial version, the value in the `Version` field is preceded by the letter `T`

Examples

Version for All Installed Products

Display version information for all installed products. The output shown here is representative. Your results may differ.

```
ver
```

```
--------------------------------------------------------------------
MATLAB Version 7.14.0.39041 (R2012a)
MATLAB License Number: 234567
Operating System: Microsoft Windows 7 Version 6.1 (Build 7600)
Java Version: Java 1.6.0_17-b04 with Sun Microsystems Inc. Java HotSpo
--------------------------------------------------------------------
MATLAB                                                  Version 7.14
Simulink                                                Version 8.0
Control System Toolbox                                  Version 9.3
```

Version for a Particular Product

Display version information for MATLAB and the Control System Toolbox product. The output shown here is representative. Your results may differ.

1. Determine the product name for Control System Toolbox by setting `n` to the name of a function unique to Control System Toolbox, such as `dss`:

```
n = 'dss';
pat = '(?<=^.+[\\/]toolbox[\\/])[^\\/]+';
regexp(which(n), pat, 'match', 'once')
```

```
ans=
control
```

2. Specify the value returned in the previous step as an argument to `ver`:

```
ver control
```

```
-----------------------------------------------------------------------
MATLAB Version 7.14.0.39041 (R2012a)
MATLAB License Number: 234567
Operating System: Microsoft Windows 7 Version 6.1 (Build 7600)
Java Version: Java 1.6.0_17-b04 with Sun Microsystems Inc. Java HotSpot(T
-----------------------------------------------------------------------
Control System Toolbox                                Version 9.3
```

Structure Containing Version for the MATLAB family of products

Create a structure containing version information, and then display the structure values. The output shown here is representative. Your results may differ.

```
v=ver;
for k = 1:length(v)
  fprintf('%s\n', v(k).Name);
  fprintf('  Version: %s\n', v(k).Version);
end
```

```
MATLAB
 Version: 7.14

MATLAB Builder JA
 Version: 2.2.4

MATLAB Builder NE
 Version: 4.2

MATLAB Compiler
 Version: 4.17
```

```
My Custom Toolbox
 Version: 1.0
```

Structure Containing Version for a Particular Product

Create a structure containing version information for just the Symbolic Math Toolbox™ product. The output shown here is representative. Your results may differ.

1. Determine the product name for Symbolic Math Toolbox by setting `n` to the name of a function unique to Symbolic Math Toolbox, such as `sym`:

```
n = 'sym';
pat = '(?<=^.+[\\/]toolbox[\\/])[^\\/]+';
regexp(which(n), pat, 'match', 'once')
```

```
ans =

symbolic
```

2. Specify the value returned in the previous step as an argument to `ver`:

```
v = ver('symbolic')
```

```
v =

       Name: 'Symbolic Math Toolbox'
    Version: '5.8'
    Release: '(R2012a)'
       Date: '14-Sep-2011'
```

See Also

`computer` | `help` | `license` | `verlessthan` | `version`

Concepts

- "Information About your Installation"
- "Create Help Summary Files (Contents.m)"

verctrl

Purpose

Source control actions (Windows platforms)

Syntax

```
verctrl('action',{'filename1','filename2',....},0)
result=verctrl('action',{'filename1','filename2',....},0)
verctrl('action','filename',0)
result=verctrl('isdiff','filename',0)
list = verctrl('all_systems')
```

Description

`verctrl('action',{'filename1','filename2',....},0)` performs the source control operation specified by `'action'` for a single file or multiple files. Enter one file as a string; specify multiple files using a cell array of strings. Use the full paths for each file name and include the extensions. Specify `0` as the last argument. Complete the resulting dialog box to execute the operation. Available values for `'action'` are as follows:

action Argument	Purpose
`'add'`	Adds files to the source control system. Files can be open in the Editor or closed when added.
`'checkin'`	Checks files into the source control system, storing the changes and creating a new version.
`'checkout'`	Retrieves files for editing.
`'get'`	Retrieves files for viewing and compiling, but not editing. When you open the files, they are labeled as read-only.
`'history'`	Displays the history of files.
`'remove'`	Removes files from the source control system. It does not delete the files from disk, but only from the source control system.

action Argument	Purpose
`'runscc'`	Starts the source control system. The file name can be an empty string.
`'uncheckout'`	Cancels a previous checkout operation and restores the contents of the selected files to the precheckout version. All changes made to the files since the checkout are lost.

`result=verctrl('`*`action`*`',{'filename1','filename2',....},0)` performs the source control operation specified by `'`*`action`*`'` on a single file or multiple files. The action can be any one of: `'add'`, `'checkin'`, `'checkout'`, `'get'`, `'history'`, or `'undocheckout'`. `result` is a logical `1` (true) when you complete the operation by clicking **OK** in the resulting dialog box, and is a logical `0` (false) when you abort the operation by clicking **Cancel** in the resulting dialog box.

`verctrl('`*`action`*`','filename',0)` performs the source control operation specified by `'`*`action`*`'` for a single file. Use the absolute path for `'filename'`. Specify `0` as the last argument. Complete any resulting dialog boxes to execute the operation. Available values for `'action'` are as follows:

action Argument	Purpose
`'showdiff'`	Displays the differences between a file and the latest checked in version of the file in the source control system.
`'properties'`	Displays the properties of a file.

`result=verctrl('`**`isdiff`**`','filename',0)` compares `filename` with the latest checked in version of the file in the source control system. `result` is a logical `1` (true) when the files are different, and is a logical `0`

(false) when the files are identical. Use the full path for `'filename'`. Specify `0` as the last argument.

`list = verctrl('all_systems')` displays in the Command Window a list of all source control systems installed on your computer.

Examples

Check In a File

Check in `D:\file1.ext` to the source control system:

```
result = verctrl('checkin','D:\file1.ext', 0)
```

This opens the Check in file(s) dialog box. Click **OK** to complete the check in. MATLAB displays

```
result = 1
```

indicating the checkin was successful.

Add Files to the Source Control System

Add `D:\file1.ext` and `D:\file2.ext` to the source control system.

```
verctrl('add',{'D:\file1.ext','D:\file2.ext'}, 0)
```

This opens the Add to source control dialog box. Click **OK** to complete the operation.

Display the Properties of a File

Display the properties of `D:\file1.ext`.

```
verctrl('properties','D:\file1.ext', 0)
```

This opens the source control properties dialog box for your source control system. The function is complete when you close the properties dialog box.

Show Differences for a File

To show the differences between the version of `file1.ext` that you just edited and saved, with the last version in source control, run

```
verctrl('showdiff','D:\file1.ext',0)
```

MATLAB displays differences dialog boxes and results specific to your source control system. After checking in the file, if you run this statement again, MATLAB displays

```
??? The file is identical to latest version under source control.
```

List All Installed Source Control Systems

To view all of the source control systems installed on your computer, type

```
list = verctrl ('all_systems')
```

MATLAB displays all the source control systems currently installed on your computer. For example:

```
list =
'Microsoft Visual SourceSafe'
'ComponentSoftware RCS'
```

See Also

checkin | checkout | undocheckout | cmopts

How To

- “Source Control Interface on Microsoft Windows”

verLessThan

Purpose

Compare toolbox version to specified version string

Syntax

```
verLessThan(toolbox, version)
```

Description

`verLessThan(toolbox, version)` returns logical 1 (`true`) if the version of the toolbox specified by the string `toolbox` is older than the version specified by the string `version`, and logical 0 (`false`) otherwise. Use this function when you want to write code that can run across multiple versions of the MATLAB software, when there are differences in the behavior of the code in the different versions.

The `toolbox` argument is a string enclosed within single quotation marks that contains the name of a MATLAB toolbox folder. The `version` argument is a string enclosed within single quotation marks that contains the version to compare against. This argument must be in the form `major[.minor[.revision]]`, such as `7`, `7.1`, or `7.0.1`. If `toolbox` does not exist, MATLAB generates an error.

To specify `toolbox`, find the folder that holds the `Contents.m` file for the toolbox and use that folder name. To see a list of all toolbox folder names, enter the following statement in the MATLAB Command Window:

```
dir([matlabroot '/toolbox'])
```

Tips

The `verLessThan` function is available with MATLAB Version 7.4 and subsequent versions. If you are running a version of MATLAB prior to 7.4, you can download the `verLessThan` function from the following MathWorks Technical Support solution. You must be running MATLAB Version 6.0 or higher to use this function:

```
http://www.mathworks.com/support/solutions/data/1-38LI61.html?solution=1-
```

Examples

These examples illustrate usage of the `verLessThan` function.

Example 1 – Checking For the Minimum Required Version

```
if verLessThan('simulink', '4.0')
    error('Simulink 4.0 or higher is required.');
```

```
end
```

Example 2 – Choosing Which Code to Run

```
if verLessThan('matlab', '7.0.1')
% -- Put code to run under MATLAB 7.0.0 and earlier here --
else
% -- Put code to run under MATLAB 7.0.1 and later here --
end
```

Example 3 – Looking Up the Folder Name

Find the name of the Data Acquisition Toolbox folder:

```
dir([matlabroot '/toolbox/d*'])

   daq        database   des      distcomp   dotnetbuilder
   dastudio   datafeed   dials    dml        dspblks
```

Use the toolbox folder name, `daq`, to compare the Data Acquisition Toolbox software version that MATLAB is currently running against version number `3`:

```
verLessThan('daq', '3')
ans =
     1
```

See Also

`ver` | `version` | `license` | `ispc` | `isunix` | `ismac` | `dir`

version

Purpose

Version number for MATLAB and libraries

Syntax

```
version
version -date
version -description
version -release
version -java
v = version('-versionOption')
[v d] = version
```

Description

`version` returns in `ans` the version and release number for the MATLAB software currently running.

`version -date` returns in `ans` the release date for the MATLAB software.

`version -description` returns in `ans` a description of the version. Usually, the description is for special versions, such as beta versions.

`version -release` returns in `ans` the release number for the MATLAB software currently running.

`version -java` returns in `ans` the version of the Sun Microsystems™ JVM software that MATLAB is using.

`v = version('-versionOption')`, where *`versionOption`* is one of the above option strings, is an alternate form of the syntax.

`[v d] = version` returns the version and release number in string `v` and the release date in string `d`. No input arguments are allowed in this syntax.

Examples

Display the version:

```
version
```

MATLAB returns:

```
7.9.0.2601 (R2009b)
```

Display the release, prefaced by a descriptor:

```
['Release R' version('-release')]
```

MATLAB returns:

```
Release R2009b
```

Return release version and release date as separate strings:

```
 [v d] = version
```

returns:

```
v =
7.12.0.39132 (R2011a)

d =
December 1, 2010
```

View the Java version:

```
version -java
```

MATLAB returns:

```
Java 1.6.0_17-b04 with Sun Microsystems Inc. Java HotSpot(TM) 64-Bit S
```

See Also

computer | ver | verlessthan

How To

- “Check for Software Updates”

Purpose

Concatenate arrays vertically

Syntax

```
C = vertcat(A1, A2, ...)
```

Description

`C = vertcat(A1, A2, ...)` vertically concatenates matrices `A1`, `A2`, and so on. All matrices in the argument list must have the same number of columns.

`vertcat` concatenates N-dimensional arrays along the first dimension. The remaining dimensions must match.

`vertcat` also concatenates character strings. Each string being concatenated must have the same number of characters.

MATLAB calls `C = vertcat(A1, A2, ...)` for the syntax `C = [A1; A2; ...]` when any of `A1`, `A2`, etc. is an object.

Tips

For information on combining unlike integer types, integers with nonintegers, cell arrays with non-cell arrays, or empty matrices with other elements, see "Valid Combinations of Unlike Classes".

Examples

Create a 5-by-3 matrix, `A`, and a 3-by-3 matrix, `B`. Then vertically concatenate `A` and `B`.

```
A = magic(5);            % Create 5-by-3 matrix, A
A(:, 4:5) = []

A =

    17    24     1
    23     5     7
     4     6    13
    10    12    19
    11    18    25


B = magic(3)*100         % Create 3-by-3 matrix, B
```

```
B =

   800   100   600
   300   500   700
   400   900   200


C = vertcat(A,B)        % Vertically concatenate A and B

C =

    17    24     1
    23     5     7
     4     6    13
    10    12    19
    11    18    25
   800   100   600
   300   500   700
   400   900   200
```

See Also

horzcat | cat

How To

- "Redefining Concatenation for Your Class"

vertcat (tscollection)

Purpose

Vertical concatenation for `tscollection` objects

Syntax

```
tsc = vertcat(tsc1,tsc2,...)
```

Description

`tsc = vertcat(tsc1,tsc2,...)` performs

```
tsc = [tsc1;tsc2;...]
```

This operation appends `tscollection` objects. The time vectors must not overlap. The last time in tsc1 must be earlier than the first time in tsc2. All `tscollection` objects to be concatenated must have the same `timeseries` members.

See Also

`horzcat (tscollection)` | `tscollection`

Purpose

(Will be removed) Return simplices attached to specified vertices

---

Note `vertexAttachments(TriRep)` will be removed in a future release. Use `vertexAttachments(triangulation)` instead.

`TriRep` will be removed in a future release. Use `triangulation` instead.

Syntax

```
SI = vertexAttachments(TR, VI)
```

Description

`SI = vertexAttachments(TR, VI)` returns the vertex-to-simplex information for the specified vertices `VI`. For 2-D triangulations in MATLAB, the triangles `SI` are arranged in counter-clockwise order around the attached vertex `VI`.

Input Arguments

`TR`	Triangulation representation
`VI`	`VI` is a column vector of indices into the array of points representing the vertex coordinates, `TR.X`. The simplices associated with vertex `i` are the `i`'th entry in the cell array. If `VI` is not specified the vertex-simplex information for the entire triangulation is returned.

Output Arguments

`SI`	Cell array of indices of the simplices attached to a vertex. A cell array is used to store the information because the number of simplices associated with each vertex can vary. The simplices associated with vertex `i` are in the `i`'th entry in the cell array `SI`.

Definitions

A simplex is a triangle/tetrahedron or higher dimensional equivalent.

Examples

Example 1

Load a 2-D triangulation and use `TriRep` to compute the vertex-to-triangle relations.

```
load trimesh2d
trep = TriRep(tri, x, y);
```

Find the indices of the tetrahedra attached to the first vertex:

```
Tv = vertexAttachments(trep, 1)
Tv{:}
```

Example 2

Perform a direct query of a 2-D triangulation created using `DelaunayTri`.

```
x = rand(20,1);
y = rand(20,1);
dt = DelaunayTri(x,y);
```

Find the triangles attached to vertex 5:

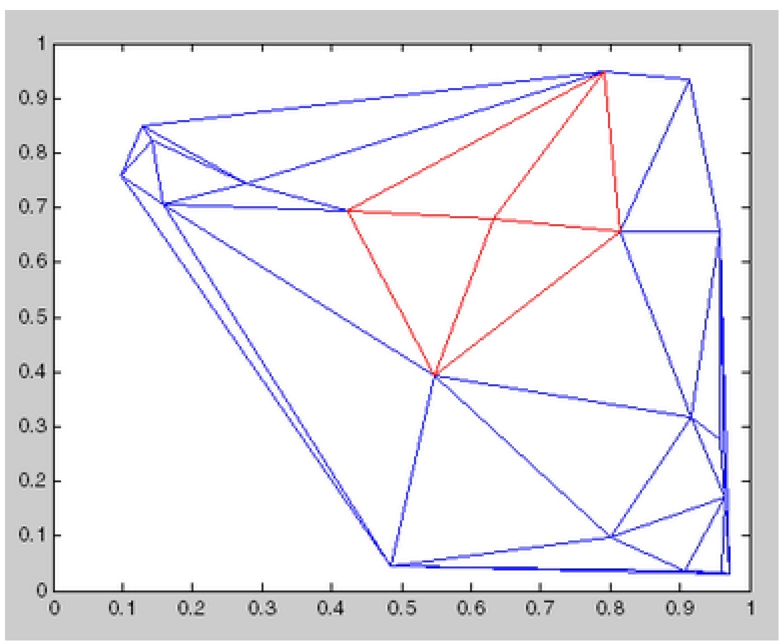
```
t = vertexAttachments(dt,5);
```

Plot the triangulation:

```
triplot(dt);
hold on;
```

Plot the triangles attached to vertex 5 (in red):

```
triplot(dt(t{:},:),x,y,'Color','r');
hold off;
```



See Also

`delaunayTriangulation` | `triangulation`

VideoReader

Purpose

Read video files

Description

Use the `VideoReader` function with the `read` method to read video data from a file into the MATLAB workspace.

The file formats that `VideoReader` supports vary by platform, as follows (with no restrictions on file extensions):

All Platforms	AVI (`.avi`), Motion JPEG 2000 (`.mj2`)
All Windows	MPEG-1 (`.mpg`), Windows Media Video (`.wmv`, `.asf`, `.asx`), and any format supported by Microsoft DirectShow®.
Windows 7	MPEG-4, including H.264 encoded video (`.mp4`, `.m4v`), Apple QuickTime Movie (`.mov`), and any format supported by Microsoft Media Foundation.
Macintosh	MPEG-1 (`.mpg`), MPEG-4, including H.264 encoded video (`.mp4`, `.m4v`), Apple QuickTime Movie (`.mov`), and any format supported by QuickTime as listed on http://support.apple.com/kb/HT3775.
Linux	Any format supported by your installed plug-ins for GStreamer 0.10 or above, as listed on http://gstreamer.freedesktop.org/documentation/plugins.html, including Ogg Theora (`.ogg`).

For more information, see “Supported Video File Formats” in the MATLAB Data Import and Export documentation.

Construction

`obj = VideoReader(filename)` constructs `obj` to read video data from the file named `filename`. If it cannot construct the object for any reason, `VideoReader` generates an error.

`obj = VideoReader(filename,Name,Value)` constructs the object with additional options specified by one or more `Name,Value` pair

arguments. `Name` is `'Tag'` or `'UserData'` and `Value` is the corresponding value. You can specify two name and value pair arguments in any order as `Name1,Value1,Name2,Value2`.

Input Arguments

filename

String in single quotation marks that specifies the video file to read. The `VideoReader` constructor searches for the file on the MATLAB path.

Name-Value Pair Arguments

Optional comma-separated pairs of `Name,Value` arguments, where `Name` is `'Tag'` or `'UserData'` and `Value` is the corresponding value. You can specify two name and value pair arguments in any order as `Name1,Value1,Name2,Value2`.

Tag

String that identifies the object.

Default: `''`

UserData

Generic field for data of any class that you want to add to the object.

Default: `[]`

Properties

All properties are read-only except `Tag` and `UserData`.

BitsPerPixel

Bits per pixel of the video data.

Duration

Total length of the file in seconds.

FrameRate

Frame rate of the video in frames per second.

Height

Height of the video frame in pixels.

Name

Name of the file associated with the object.

NumberOfFrames

Total number of frames in the video stream.

Some files store video at a variable frame rate, including many Windows Media Video files. For these files, `VideoReader` cannot determine the number of frames until you read the last frame. When you construct the object, `VideoReader` returns a warning and does not set the `NumberOfFrames` property.

To count the number of frames in a variable frame rate file, use the `read` method to read the last frame of the file. For example:

```
vidObj = VideoReader('varFrameRateFile.wmv');
lastFrame = read(vidObj, inf);
numFrames = vidObj.NumberOfFrames;
```

For more information, see "Reading Variable Frame Rate Video" in the MATLAB Data Import and Export documentation..

Path

String containing the full path to the file associated with the reader.

Tag

String that identifies the object.

Default: `''`

Type

Class name of the object: `'VideoReader'`.

UserData

Generic field for data of any class that you want to add to the object.

Default: `[]`

VideoFormat

String indicating the MATLAB representation of the video format.

Video Format	Value of `VideoFormat`
AVI or MPEG-4 files with RGB24 video	`'RGB24'`
AVI files with indexed video	`'Indexed'`
AVI files with grayscale video	`'Grayscale'`

For Motion JPEG 2000 files, `VideoFormat` is one of the following.

Format of Image Data	Value of `VideoFormat`
Single-band `uint8`	`'Mono8'`
Single-band `int8`	`'Mono8 Signed'`
Single-band `uint16`	`'Mono16'`
Single-band `int16`	`'Mono16 Signed'`
Three-banded `uint8`	`'RGB24'`
Three-banded `int8`	`'RGB24 Signed'`
Three-banded `uint16`	`'RGB48'`
Three-banded `int16`	`'RGB48 Signed'`

Width

Width of the video frame in pixels.

Methods

get	Query property values for video reader object
getFileFormats	File formats that `VideoReader` supports
read	Read video frame data from file
set	Set property values for video reader object

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Construct a `VideoReader` object for the example movie file `xylophone.mpg` and view its properties:

```
xyloObj = VideoReader('xylophone.mpg', 'Tag', 'My reader object');
get(xyloObj)
```

Read and play back the movie file `xylophone.mpg`:

```
xyloObj = VideoReader('xylophone.mpg');

nFrames = xyloObj.NumberOfFrames;
vidHeight = xyloObj.Height;
vidWidth = xyloObj.Width;

% Preallocate movie structure.
mov(1:nFrames) = ...
    struct('cdata', zeros(vidHeight, vidWidth, 3, 'uint8'),...
           'colormap', []);

% Read one frame at a time.
```

```
for k = 1 : nFrames
    mov(k).cdata = read(xyloObj, k);
end

% Size a figure based on the video's width and height.
hf = figure;
set(hf, 'position', [150 150 vidWidth vidHeight])

% Play back the movie once at the video's frame rate.
movie(hf, mov, 1, xyloObj.FrameRate);
```

See Also

mmfileinfo | VideoWriter

How To

- "Read Video Files"

Purpose

Write video files

Description

Use the `VideoWriter` object with the `open`, `writeVideo`, and `close` methods to create video files from figures, still images, or MATLAB movies. `VideoWriter` can create uncompressed AVI and Motion JPEG 2000 compressed AVI files on all platforms, and MPEG-4 files on Windows 7 and Mac OS X 10.7 and higher. `VideoWriter` supports files larger than 2 GB, which is the limit for `avifile`.

To set video properties, `VideoWriter` includes predefined *profiles* such as `'Uncompressed AVI'` or `'MPEG-4'`.

Construction

`writerObj = VideoWriter(filename)` constructs a `VideoWriter` object to write video data to an AVI file with Motion JPEG compression.

`writerObj = VideoWriter(filename,profile)` applies a set of properties tailored to a specific file format (such as `'MPEG-4'` or `'Uncompressed AVI'`) to a `VideoWriter` object.

Input Arguments

filename

String enclosed in single quotation marks that specifies the name of the file to create.

`VideoWriter` supports these file extensions:

`.avi`	AVI file
`.mj2`	Motion JPEG 2000 file
`.mp4` or `.m4v`	MPEG-4 file (systems with Windows 7 or Mac OS X 10.7 and later)

If you do not specify a valid file extension, `VideoWriter` appends the extension `.avi`, `.mj2` or `.mp4`, depending on the `profile`. If you do not specify a value for `profile`, then `VideoWriter` creates a Motion JPEG compressed AVI file with the extension `.avi`.

profile

String enclosed in single quotation marks that describes the type of file to create. Specifying a profile sets default values for video properties such as `VideoCompressionMethod`. Possible values:

`'Archival'`	Motion JPEG 2000 file with lossless compression
`'Motion JPEG AVI'`	Compressed AVI file using Motion JPEG codec
`'Motion JPEG 2000'`	Compressed Motion JPEG 2000 file
`'MPEG-4'`	Compressed MPEG-4 file with H.264 encoding (systems with Windows 7 or Mac OS X 10.7 and later)
`'Uncompressed AVI'`	Uncompressed AVI file with `RGB24` video
`'Indexed AVI'`	Uncompressed AVI file with indexed video
`'Grayscale AVI'`	Uncompressed AVI file with grayscale video

Default: `'Motion JPEG AVI'`

Properties

ColorChannels

Number of color channels in each output video frame. (Read-only)

AVI and MPEG-4 files with RGB24 data have three color channels. Indexed and Grayscale AVI files have one color channel. The number of channels for Motion JPEG 2000 files depends on the input data to the `writeVideo` method: one for monochrome image data, three for color data.

Colormap

P-by-3 numeric matrix that contains color information about the video file. The colormap can have a maximum of 256 entries of type uint8 or double. The entries of the colormap must be integers. Each row of Colormap specifies the red, green, and blue components of a single color. The colormap can be set explicitly before the call to open, or using the colormap field of a movie frame structure at the time of writing the first frame.

Only applies to objects associated with Indexed AVI files.

After you call open, you cannot change the Colormap value.

Default: none

CompressionRatio

Number greater than 1 that specifies the target ratio between the number of bytes in the input image and the number of bytes in the compressed image. The data is compressed as much as possible, up to the specified target.

Only available for objects associated with Motion JPEG 2000 files. After you call open, you cannot change the CompressionRatio value. If you previously set LosslessCompression to true, setting CompressionRatio generates an error.

Default: 10

Duration

Scalar value specifying the duration of the file in seconds. (Read-only)

FileFormat

String specifying the type of file to write: 'avi', 'mp4', or 'mj2'. (Read-only)

Filename

String specifying the name of the file. (Read-only)

FrameCount

Number of frames written to the video file. (Read-only)

FrameRate

Rate of playback for the video in frames per second. After you call `open`, you cannot change the `FrameRate` value.

Default: `30`

Height

Height of each video frame in pixels. The `writeVideo` method sets values for `Height` and `Width` based on the dimensions of the first frame. (Read-only)

MPEG-4 files require frame dimensions that are divisible by two. If the input frame height for an MPEG-4 file is not an even number, `VideoWriter` pads the frame with a row of black pixels at the bottom.

LosslessCompression

Boolean value (logical `true` or `false`) only available for objects associated with Motion JPEG 2000 files. If `true`:

- The `writeVideo` method uses reversible mode so that the decompressed data is identical to the input data.
- `VideoWriter` ignores any specified value for `CompressionRatio`.

After you call `open`, you cannot change the `LosslessCompression` value.

Default: `false` for the `'Motion JPEG 2000'` profile, `true` for the `'Archival'` profile

MJ2BitDepth

Number of least significant bits in the input image data, from 1 to 16.

Only available for objects associated with Motion JPEG 2000 files. If you do not specify a value before calling the `open` method, `VideoWriter` sets the bit depth based on the input data type. For example, if the input data to `writeVideo` is an array of `uint8` or `int8` values, `MJ2BitDepth` is `8`.

Path

String specifying the fully qualified path. (Read-only)

Quality

Integer from 0 through 100. Higher quality numbers result in higher video quality and larger file sizes. Lower quality numbers result in lower video quality and smaller file sizes.

Only available for objects associated with the `MPEG-4` or `Motion JPEG AVI` profile. After you call `open`, you cannot change the `Quality` value.

Default: `75`

VideoBitsPerPixel

Number of bits per pixel in each output video frame. (Read-only)

AVI and MPEG-4 files have 24 bits per pixel (8 bits for each of three color bands).

For Motion JPEG 2000 files, the number of bits per pixel depends on the value of `MJ2BitDepth` and the number of bands of image data. For example, if the input data to `writeVideo` is a three-dimensional array of `uint16` or `int16` values, the default value of `MJ2BitDepth` is `16`, and `VideoBitsPerPixel` is `48` (three times the bit depth).

VideoCompressionMethod

String indicating the type of video compression: 'None', 'H.264', 'Motion JPEG', or 'Motion JPEG 2000'. (Read-only)

VideoFormat

String indicating the MATLAB representation of the video format. (Read-only)

Video Format	Value of VideoFormat
AVI or MPEG-4 files with RGB24 video	'RGB24'
AVI files with indexed video	'Indexed'
AVI files with grayscale video	'Grayscale'

For Motion JPEG 2000 files, VideoFormat depends on the value of MJ2BitDepth and the format of the input image data to the writeVideo method. For example, if you do not specify the MJ2BitDepth property, VideoWriter sets the format as shown in this table.

Format of Image Data	Value of VideoFormat
Single-band uint8	'Mono8'
Single-band int8	'Mono8 Signed'
Single-band uint16	'Mono16'
Single-band int16	'Mono16 Signed'
Three-banded uint8	'RGB24'
Three-banded int8	'RGB24 Signed'
Three-banded uint16	'RGB48'
Three-banded int16	'RGB48 Signed'

Width

Width of each video frame in pixels. The `writeVideo` method sets values for `Height` and `Width` based on the dimensions of the first frame. (Read-only)

MPEG-4 files require frame dimensions that are divisible by two. If the input frame width for an MPEG-4 file is not an even number, `VideoWriter` pads the frame with a column of black pixels along the right side.

Methods

close	Close file after writing video data
getProfiles	List profiles and file formats supported by `VideoWriter`
open	Open file for writing video data
writeVideo	Write video data to file

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

View and Set Object Properties

Construct a `VideoWriter` object and view its properties.

```
writerObj = VideoWriter('newfile.avi')
```

Set the frame rate to 60 frames per second, using the `FrameRate` property.

```
writerObj.FrameRate = 60;
```

AVI File from Animation

Write a sequence of frames to a compressed AVI file, `peaks.avi`.

Prepare the new file.

```
writerObj = VideoWriter('peaks.avi');
```

```
open(writerObj);
```

Generate initial data and set axes and figure properties.

```
Z = peaks; surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
set(gcf,'Renderer','zbuffer');
```

Setting the `Renderer` property to `zbuffer` or `Painters` works around limitations of `getframe` with the OpenGL renderer on some Windows systems.

Create a set of frames and write each frame to the file.

```
for k = 1:20
   surf(sin(2*pi*k/20)*Z,Z)
   frame = getframe;
   writeVideo(writerObj,frame);
end

close(writerObj);
```

See Also

`VideoReader` | `mmfileinfo`

view

Purpose

Viewpoint specification

Syntax

```
view(az,el)
view([az,el])
view([x,y,z])
view(2)
view(3)
view(ax,...)
[az,el] = view
```

Description

The position of the viewer (the viewpoint) determines the orientation of the axes. You specify the viewpoint in terms of azimuth and elevation, or by a point in three-dimensional space.

`view(az,el)` and `view([az,el])` set the viewing angle for a three-dimensional plot. The azimuth, `az`, is the horizontal rotation about the *z*-axis as measured in degrees from the negative *y*-axis. Positive values indicate counterclockwise rotation of the viewpoint. `el` is the vertical elevation of the viewpoint in degrees. Positive values of elevation correspond to moving above the object; negative values correspond to moving below the object.

`view([x,y,z])` sets the viewpoint to the Cartesian coordinates `x`, `y`, and `z`. The magnitude of `(x,y,z)` is ignored.

`view(2)` sets the default two-dimensional view, `az = 0, el = 90`.

`view(3)` sets the default three-dimensional view, `az =  37.5, el = 30`.

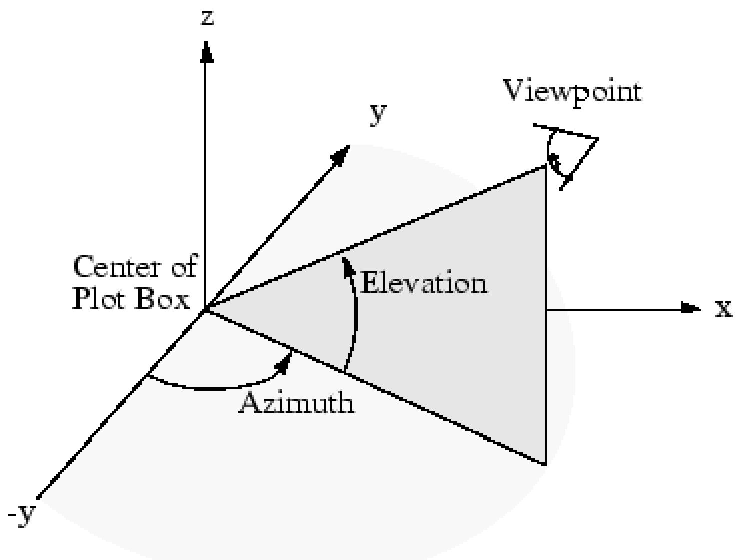
`view(ax,...)` uses axes `ax` instead of the current axes.

`[az,el] = view` returns the current azimuth and elevation.

Tips

Azimuth is a polar angle in the *x*-*y* plane, with positive angles indicating counterclockwise rotation of the viewpoint. Elevation is the angle above (positive angle) or below (negative angle) the *x*-*y* plane.

This diagram illustrates the coordinate system. The arrows indicate positive directions.



Examples

View the object from directly overhead.

```
az = 0;
el = 90;
view(az, el);
```

Set the view along the *y*-axis, with the *x*-axis extending horizontally and the *z*-axis extending vertically in the figure.

```
view([0 0]);
```

Rotate the view about the *z*-axis by 180º.

```
az = 180;
el = 90;
view(az, el);
```

See Also

`hgtransform` | `rotate3d` | CameraPosition | CameraTarget | CameraViewAngle | Projection

How To

- "View Overview"
-

Purpose

View transformation matrices

Syntax

```
viewmtx
T = viewmtx(az,el)
T = viewmtx(az,el,phi)
T = viewmtx(az,el,phi,xc)
```

Description

`viewmtx` computes a 4-by-4 orthographic or perspective transformation matrix that projects four-dimensional homogeneous vectors onto a two-dimensional view surface (e.g., your computer screen).

`T = viewmtx(az,el)` returns an *orthographic* transformation matrix corresponding to azimuth `az` and elevation `el`. `az` is the azimuth (i.e., horizontal rotation) of the viewpoint in degrees. `el` is the elevation of the viewpoint in degrees. This returns the same matrix as the commands

```
view(az,el)
T = view
```

but does not change the current view.

`T = viewmtx(az,el,phi)` returns a *perspective* transformation matrix. `phi` is the perspective viewing angle in degrees. `phi` is the subtended view angle of the normalized plot cube (in degrees) and controls the amount of perspective distortion.

Phi	Description
0 degrees	Orthographic projection
10 degrees	Similar to telephoto lens
25 degrees	Similar to normal lens
60 degrees	Similar to wide-angle lens

`T = viewmtx(az,el,phi,xc)` returns the perspective transformation matrix using `xc` as the target point within the normalized plot cube (i.e., the camera is looking at the point `xc`). `xc` is the target point that is the

center of the view. You specify the point as a three-element vector, `xc = [xc,yc,zc]`, in the interval [0,1]. The default value is `xc = [0,0,0]`.

A four-dimensional homogenous vector is formed by appending a 1 to the corresponding three-dimensional vector. For example, `[x,y,z,1]` is the four-dimensional vector corresponding to the three-dimensional point `[x,y,z]`.

Examples

Determine the projected two-dimensional vector corresponding to the three-dimensional point (0.5,0.0,-3.0) using the default view direction. Note that the point is a column vector.

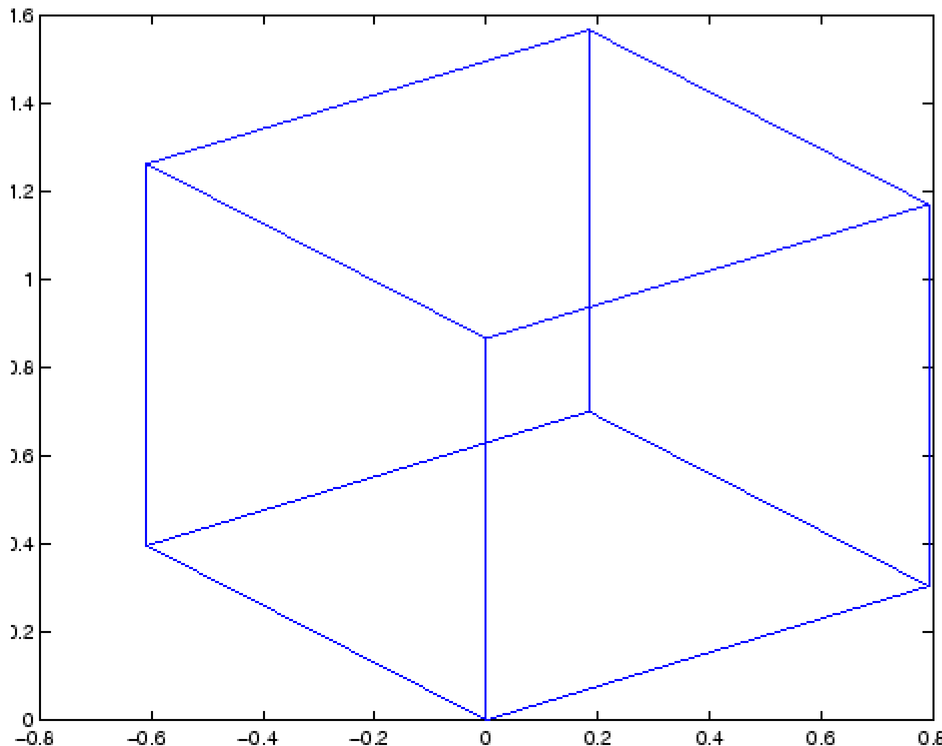
```
A = viewmtx(-37.5,30);
x4d = [.5 0 -3 1]';
x2d = A*x4d;
x2d = x2d(1:2)
x2d =
   0.3967
   -2.4459
```

These vectors trace the edges of a unit cube:

```
x = [0  1  1  0  0  0  1  1  0  0  1  1  1  1  0  0];
y = [0  0  1  1  0  0  0  1  1  0  0  0  1  1  1  1];
z = [0  0  0  0  0  1  1  1  1  1  1  0  0  1  1  0];
```

Transform the points in these vectors to the screen, then plot the object.

```
A = viewmtx(-37.5,30);
[m,n] = size(x);
x4d = [x(:),y(:),z(:),ones(m*n,1)]';
x2d = A*x4d;
x2 = zeros(m,n); y2 = zeros(m,n);
x2(:) = x2d(1,:);
y2(:) = x2d(2,:);
figure
plot(x2,y2)
```



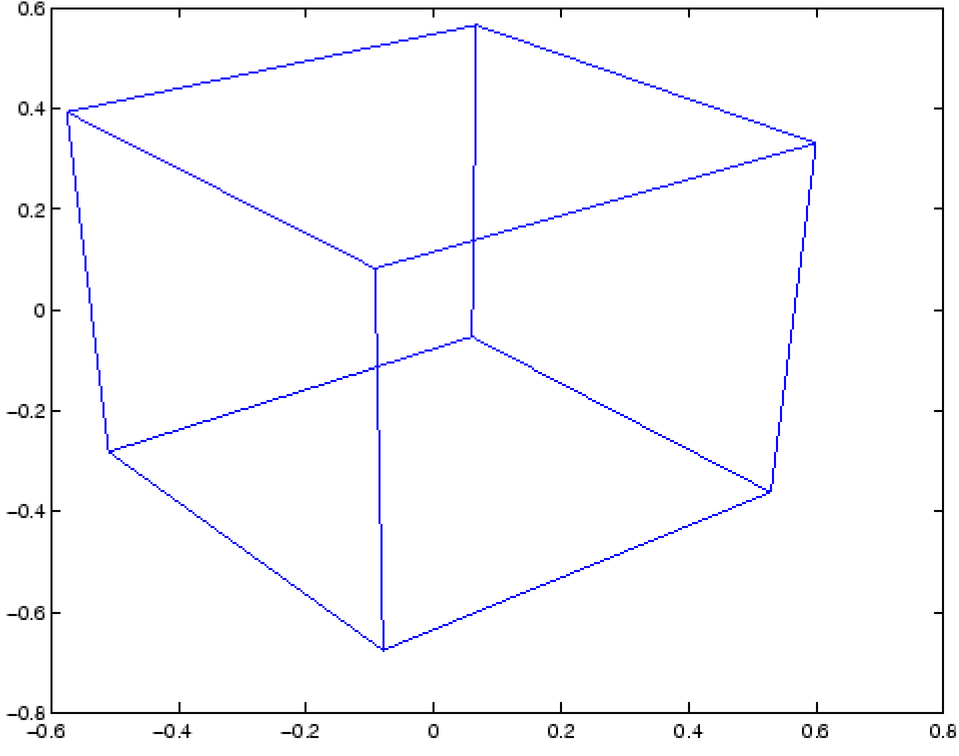
Use a perspective transformation with a 25 degree viewing angle:

```
A = viewmtx(-37.5,30,25);
x4d = [.5 0 -3 1]';
x2d = A*x4d;
x2d = x2d(1:2)/x2d(4) % Normalize
x2d =
   0.1777
   -1.8858
```

Transform the cube vectors to the screen and plot the object:

```
figure
A = viewmtx(-37.5,30,25);
```

```
[m,n] = size(x);
x4d = [x(:),y(:),z(:),ones(m*n,1)]';
x2d = A*x4d;
x2 = zeros(m,n); y2 = zeros(m,n);
x2(:) = x2d(1,:)./x2d(4,:);
y2(:) = x2d(2,:)./x2d(4,:);
plot(x2,y2)
```



See Also

`view` | `hgtransform`

Tutorials

- “Defining Scenes with Cameras”

Purpose

Compare two text files, MAT-Files, binary files, Zip files, or folders

Syntax

```
visdiff('fname1', 'fname2')
visdiff('filename1', 'filename2', 'type')
```

Description

`visdiff('fname1', 'fname2')` opens the Comparison Tool and presents the differences between the two files or folders. Either ensure that the two files or folders appear on the MATLAB path, or provide the full path for each file or folder. You can compare files or any combination of folders, zip files, or Simulink manifests.

If you have Simulink Report Generator™ software, you can select a pair of Simulink models to compare XML text files generated from them.

`visdiff('filename1', 'filename2', 'type')` opens the Comparison Tool and presents the differences between the two files using the specified comparison type. *`type`* can be `'text'` or `'binary'`. If you do not specify `type`, `visdiff` creates the default comparison type for your selected files. The *`type`* option does not apply when comparing folders.

If there are multiple comparison types available for your selections (e.g., text, binary, file list, or XML comparison), you can create a new comparison and choose a different comparison type.

1. In the Comparison Tool, click the New comparison button.

   The dialog box Select Files or Folders for Comparison opens and shows your previous comparison selections in the drop-down lists.

2. Change the comparison type and click **Compare**.

Examples

Specifying Files or Folders to Compare

The `visdiff` function accepts fully qualified file names, relative file names, or names of files on the MATLAB path.

If the files you want to compare appear on the MATLAB path or in the current folder, you can specify the file names without the full path, for example:

```
visdiff('lengthofline.m','lengthofline2.m')
```

or

```
visdiff('lengthofline','lengthofline2')
```

If the files you want to compare are not on the path, either specify the full path to each file, or add the folders to the path.

For example, to specify the fully qualified file names to compare two example files:

```
visdiff(fullfile(matlabroot,'toolbox','matlab','demos','gatlin.mat'), ...
fullfile(matlabroot,'toolbox','matlab','demos','gatlin2.mat'))
```

Specify the full path to files as follows:

```
visdiff('C:\Work\comp\lengthofline.m', 'C:\Work\comp\lengthofline2.m')
```

You can specify paths to files relative to the current folder. For the preceding example, if the current folder is `Work`, then the relative paths are:

```
visdiff('comp\lengthofline.m', 'comp\lengthofline2.m')
```

Compare Two Text Files

To view a comparison of the two example files, `lengthofline.m` and `lengthofline2.m`:

```
visdiff(fullfile(matlabroot,'help','techdoc','matlab_env',...
'examples','lengthofline.m'), fullfile(matlabroot,'help',...
```

```
'techdoc','matlab_env','examples','lengthofline2.m'))
```

For information about using the report features, see “Comparing Text Files”.

Note If the text files you compare are XML files, you see different results if you have MATLAB Report Generator installed. For details, see “Comparing Files and Folders”.

Compare Two MAT-Files

To compare two example files:

```
visdiff(fullfile(matlabroot,'toolbox','matlab','demos','gatlin.mat'), ...
fullfile(matlabroot,'toolbox','matlab','demos','gatlin2.mat'))
```

For information about the report features, see “Comparing MAT-Files”.

Compare Two Binary Files

The following example code adds a folder containing two MEX-files to the MATLAB path, and then compares the files:

```
addpath([matlabroot '\extern\examples\shrlib'])
visdiff('shrlibsample.mexw32', 'yprime.mexw32')
```

The Comparison Tool opens and indicates that the files are different, but does not provide details about the differences.

For more information on binary comparisons, see “Comparing Binary Files”.

Compare Two Folders or Zip Files

You can perform file list comparisons for any combinations of folders and zip files. To view an example folder comparison and instructions for using the report features, see “Comparing Folders and Zip Files”.

Compare Files and Specify Type

To compare two example text files and specify comparison type as binary:

```
visdiff(fullfile(matlabroot,'help','techdoc','matlab_env',...
'examples','lengthofline.m'), fullfile(matlabroot,'help',...
'techdoc','matlab_env','examples','lengthofline2.m'), 'binary')
```

If you do not specify type, `visdiff` creates the default comparison type for your selected files, in this case, text comparison. By changing to the binary comparison type you could examine differences such as end-of-line characters.

Similarly, when you compare XML files without specifying type, you get a hierarchical XML comparison report. If instead you want a text or binary comparison, you can specify "text" or "binary" comparison types to see more details. When you compare zip files, the default comparison type is a file list comparison, and you might want to specify a binary comparison instead.

Alternatives

As an alternative to the `visdiff` function, compare files and folders using any of these GUI methods:

- From the Current Folder browser:
  - Select a file or folder. Right-click the file or folder, and select **Compare Against**.
  - For two files or subfolders in the same folder, select the files or folders. Then, right-click, and select **Compare Selected Files/Folders**.
- From the MATLAB desktop, on the **Home** tab, in the **File** section, click **Compare** and then select the files or folders to compare.
- If you have a file open in the Editor, on the **Editor** tab, in the **File** section, click **Compare**. Alternatively, under **Compare**, you can choose a file to compare against, or compare with the autosave

version or the version on disk. See “Comparing Files with Autosave Version or Version on Disk”.

How To

- “Comparing Files and Folders”

volumebounds

Purpose

Coordinate and color limits for volume data

Syntax

```
lims = volumebounds(X,Y,Z,V)
lims = volumebounds(X,Y,Z,U,V,W)
lims = volumebounds(V), lims = volumebounds(U,V,W)
```

Description

`lims = volumebounds(X,Y,Z,V)` returns the x, y, z, and color limits of the current axes for scalar data. `lims` is returned as a vector:

```
[xmin xmax ymin ymax zmin zmax cmin cmax]
```

You can pass this vector to the `axis` command.

`lims = volumebounds(X,Y,Z,U,V,W)` returns the x, y, and z limits of the current axes for vector data. `lims` is returned as a vector:

```
[xmin xmax ymin ymax zmin zmax]
```

`lims = volumebounds(V), lims = volumebounds(U,V,W)` assumes `X`, `Y`, and `Z` are determined by the expression

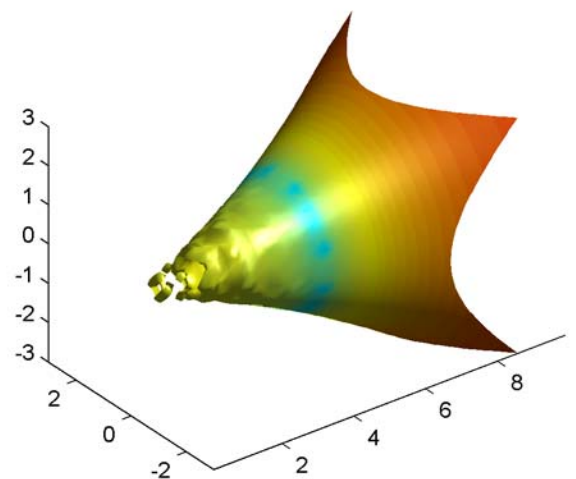
```
[X Y Z] = meshgrid(1:n,1:m,1:p)
```

where `[m n p] = size(V)`.

Examples

This example uses `volumebounds` to set the axis and color limits for an `isosurface` generated by the `flow` function.

```
[x y z v] = flow;
p = patch(isosurface(x,y,z,v,-3));
isonormals(x,y,z,v,p)
daspect([1 1 1])
isocolors(x,y,z,flipdim(v,2),p)
shading interp
axis(volumebounds(x,y,z,v))
view(3)
camlight
lighting phong
```



See Also

`isosurface` | `streamslice`

Purpose

Voronoi diagram

Note Qhull-specific options are no longer supported. Remove the `OPTIONS` argument from all instances in your code that pass it to `voronoi`.

Syntax

```
voronoi(x,y)
voronoi(x,y,TRI)
voronoi(dt)
voronoi(AX,...)
voronoi(...,'LineSpec')
h = voronoi(...)
[vx,vy] = voronoi(...)
```

Description

`voronoi(x,y)` plots the bounded cells of the Voronoi diagram for the points `x,y`. Lines-to-infinity are approximated with an arbitrarily distant endpoint.

`voronoi(x,y,TRI)` uses the triangulation `TRI` instead of computing internally.

`voronoi(dt)` uses the Delaunay triangulation `dt` instead of computing it.

`voronoi(AX,...)` plots into `AX` instead of `gca`.

`voronoi(...,'LineSpec')` plots the diagram with color and line style specified.

`h = voronoi(...)` returns, in `h`, handles to the line objects created.

`[vx,vy] = voronoi(...)` returns the finite vertices of the Voronoi edges in `vx` and `vy`.

Note For the topology of the Voronoi diagram, i.e., the vertices for each Voronoi cell, use `voronoin`.

```
[v,c] = voronoin([x(:) y(:)])
```

Definitions

Consider a set of coplanar points P. For each point P_x in the set P, you can draw a boundary enclosing all the intermediate points lying closer to P_x than to other points in the set P. Such a boundary is called a *Voronoi polygon*, and the set of all Voronoi polygons for a given point set is called a *Voronoi diagram*.

Visualization

Use one of these methods to plot a Voronoi diagram:

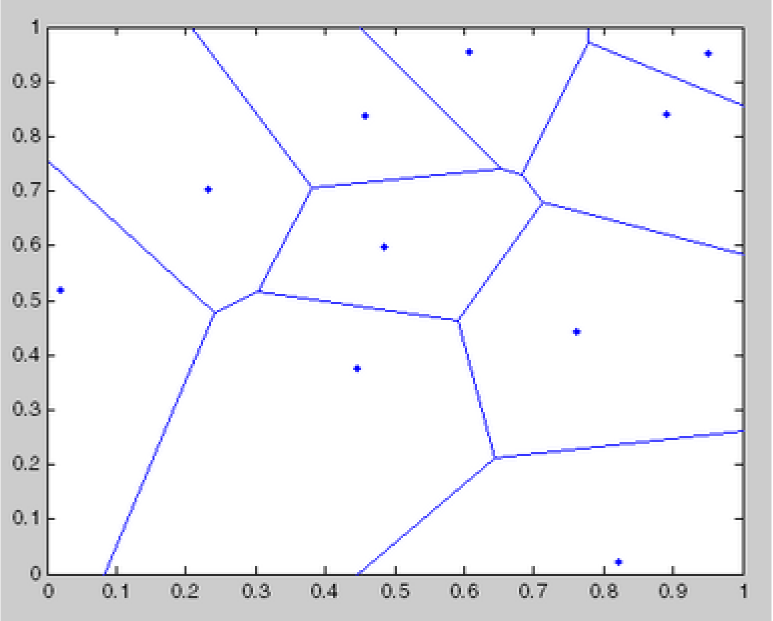
- If you provide no output argument, `voronoi` plots the diagram.
- To gain more control over color, line style, and other figure properties, use the syntax `[vx,vy] = voronoi(...)`. This syntax returns the vertices of the finite Voronoi edges, which you can then plot with the `plot` function.
- To fill the cells with color, use `voronoin` with `n = 2` to get the indices of each cell, and then use `patch` and other plot functions to generate the figure. Note that `patch` does not fill unbounded cells with color.

Examples

Example 1

This code uses the `voronoi` function to plot the Voronoi diagram for 10 randomly generated points.

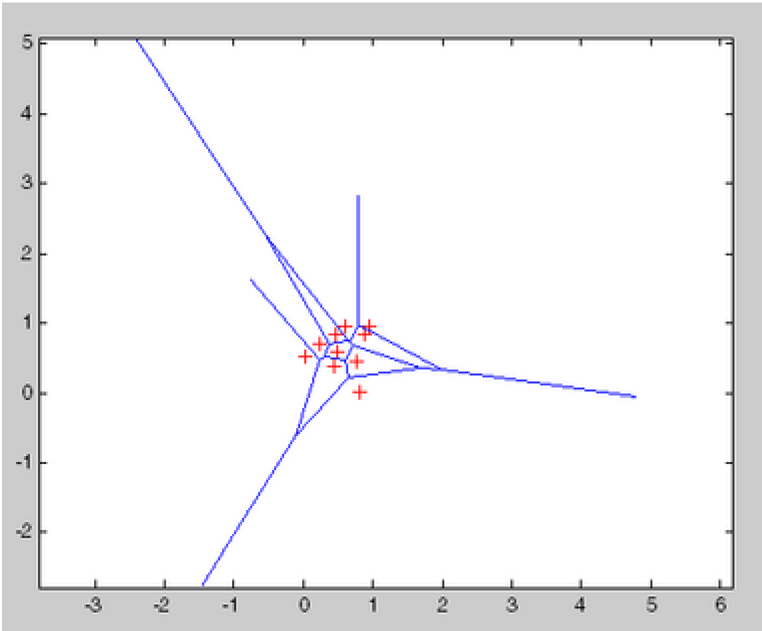
```
x = gallery('uniformdata',[1 10],0);
y = gallery('uniformdata',[1 10],1);
voronoi(x,y)
```



Example 2

This code uses the vertices of the finite Voronoi edges to plot the Voronoi diagram for the same 10 points.

```
x = gallery('uniformdata',[1 10],0);
y = gallery('uniformdata',[1 10],1);
[vx, vy] = voronoi(x,y);
plot(x,y,'r+',vx,vy,'b-'); axis equal
```



Note that you can add this code to get the figure shown in Example 1.

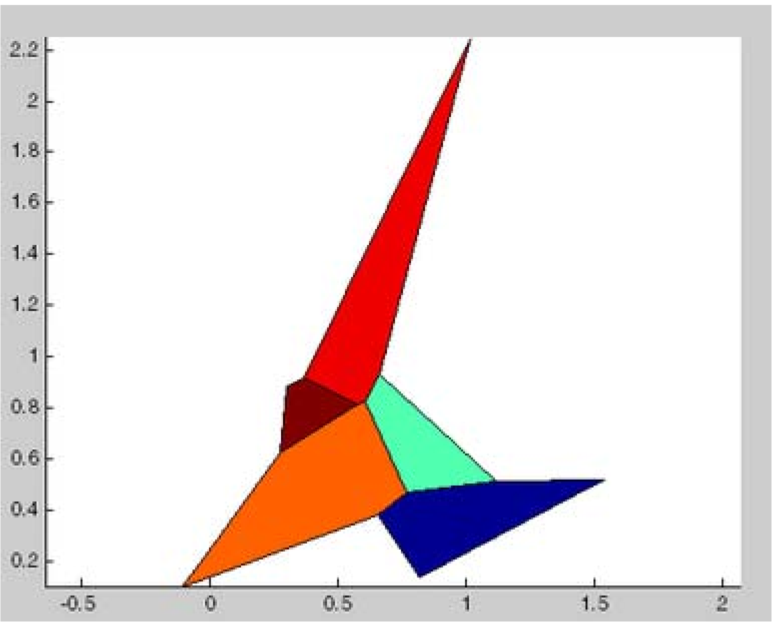
```
xlim([min(x) max(x)])
ylim([min(y) max(y)])
```

Example 3

This code uses `voronoin` and `patch` to fill the bounded cells of the same Voronoi diagram with color.

```
x = gallery('uniformdata',[10 2],5);
[v,c]=voronoin(x);
for i = 1:length(c)
if all(c{i}~=1)   % If at least one of the indices is 1,
                  % then it is an open region and we can't
                  % patch that.
patch(v(c{i},1),v(c{i},2),i); % use color i.
end
```

```
end
```



See Also

delaunayTriangulation | convhull | delaunay | LineSpec | plot | voronoin

Purpose

(Will be removed) Voronoi diagram

Note voronoiDiagram(DelaunayTri) will be removed in a future release. Use voronoiDiagram(delaunayTriangulation) instead.

DelaunayTri will be removed in a future release. Use delaunayTriangulation instead.

Syntax

```
[V, R] = voronoiDiagram(DT)
```

Description

[V, R] = voronoiDiagram(DT) returns the vertices V and regions R of the Voronoi diagram of the points DT.X. The region R{i} is a cell array of indices into V that represents the Voronoi vertices bounding the region. The Voronoi region associated with the i'th point, DT.X(i) is R{i}. For 2-D, vertices in R{i} are listed in adjacent order, i.e. connecting them will generate a closed polygon (Voronoi diagram). For 3-D the vertices in R{i} are listed in ascending order.

The Voronoi regions associated with points that lie on the convex hull of DT.X are unbounded. Bounding edges of these regions radiate to infinity. The vertex at infinity is represented by the first vertex in V.

Input Arguments

DT	Delaunay triangulation.

Output Arguments

`V`	`numv`-by-`ndim` matrix representing the coordinates of the Voronoi vertices, where `numv` is the number of vertices and `ndim` is the dimension of the space where the points reside.
`R`	Vector cell array of `length(DR.X)`, representing the Voronoi cell associated with each point.

Definitions

The *Voronoi diagram* of a discrete set of points `X` decomposes the space around each point `X(i)` into a region of influence `R{i}`. Locations within the region are closer to point `i` than any other point. The region of influence is called the Voronoi region. The collection of all the Voronoi regions is the Voronoi diagram.

The *convex hull* of a set of points `X` is the smallest convex polygon (or polyhedron in higher dimensions) containing all of the points of `X`.

Examples

Compute the Voronoi Diagram of a set of points:

```
X = [ 0.5    0
      0      0.5
     -0.5   -0.5
     -0.2   -0.1
     -0.1    0.1
      0.1   -0.1
      0.1    0.1 ]
dt = DelaunayTri(X)
[V,R] = voronoiDiagram(dt)
```

See Also

`voronoi` | `voronoin` | `triangulation` | `delaunayTriangulation`

Purpose

N-D Voronoi diagram

Syntax

```
[V,C] = voronoin(X)
[V,C] = voronoin(X,options)
```

Description

`[V,C] = voronoin(X)` returns Voronoi vertices `V` and the Voronoi cells `C` of the Voronoi diagram of `X`. `V` is a `numv`-by-`n` array of the `numv` Voronoi vertices in n-dimensional space, each row corresponds to a Voronoi vertex. `C` is a vector cell array where each element contains the indices into `V` of the vertices of the corresponding Voronoi cell. `X` is an `m`-by-`n` array, representing `m` n-dimensional points, where `n > 1` and `m >= n+1`.

The first row of `V` is a point at infinity. If any index in a cell of the cell array is `1`, then the corresponding Voronoi cell contains the first point in `V`, a point at infinity. This means the Voronoi cell is unbounded.

`voronoin` uses Qhull.

`[V,C] = voronoin(X,options)` specifies a cell array of strings `options` to be used in Qhull. The default options are

- `{'Qbb'}` for 2- and 3-dimensional input
- `{'Qbb','Qx'}` for 4 and higher-dimensional input

If `options` is `[]`, the default options are used. If `code` is `{''}`, no options are used, not even the default. For more information on Qhull and its options, see `http://www.qhull.org`.

Visualization

You can plot individual bounded cells of an n-dimensional Voronoi diagram. To do this, use `convhulln` to compute the vertices of the facets that make up the Voronoi cell. Then use `patch` and other plot functions to generate the figure.

Examples

Example 1

Let

```
x = [ 0.5    0
      0      0.5
```

```
     -0.5   -0.5
     -0.2   -0.1
     -0.1    0.1
      0.1   -0.1
      0.1    0.1 ]
```

then

```
[V,C] = voronoin(x)

V =
        Inf       Inf
        0.3833    0.3833
        0.7000   -1.6500
        0.2875    0.0000
       -0.0000    0.2875
       -0.0000   -0.0000
       -0.0500   -0.5250
       -0.0500   -0.0500
       -1.7500    0.7500
       -1.4500    0.6500
C =

    [1x4 double]
    [1x5 double]
    [1x4 double]
    [1x4 double]
    [1x4 double]
    [1x5 double]
    [1x4 double]
```

Use a `for` loop to see the contents of the cell array `C`.

```
for i=1:length(C), disp(C{i}), end

     4     2     1     3
    10     5     2     1     9
     9     1     3     7
```

```
    10     8     7     9
    10     5     6     8
     8     6     4     3     7
     6     4     2     5
```

In particular, the fifth Voronoi cell consists of 4 points: `V(10,:)`, `V(5,:)`, `V(6,:)`, `V(8,:)`.

Example 2

The following example illustrates the `options` input to `voronoin`. The commands

```
X = [-1 -1; 1 -1; 1 1; -1 1];
[V,C] = voronoin(X)
```

return an error message.

```
? qhull input error: can not scale last coordinate. Input is
cocircular
   or cospherical. Use option 'Qz' to add a point at infinity.
```

The error message indicates that you should add the option `'Qz'`. The following command passes the option `'Qz'`, along with the default `'Qbb'`, to `voronoin`.

```
[V,C] = voronoin(X,{'Qbb','Qz'})
V =

   Inf   Inf
     0     0


C =

    [1x2 double]
    [1x2 double]
    [1x2 double]
    [1x2 double]
```

Algorithms

`voronoin` is based on Qhull [1]. For information about Qhull, see `http://www.qhull.org/`.

References

[1] Barber, C. B., D.P. Dobkin, and H.T. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," *ACM Transactions on Mathematical Software*, Vol. 22, No. 4, Dec. 1996, p. 469-483.

See Also

`delaunayTriangulation` | `convhull` | `convhulln` | `delaunay` | `delaunayn` | `voronoi`

Purpose

Wait until timer stops running

Syntax

```
wait(obj)
```

Description

`wait(obj)` blocks the MATLAB command line and waits until the timer, represented by the timer object `obj`, stops running. When a timer stops running, the value of the timer object's `Running` property changes from `'on'` to `'off'`.

If `obj` is an array of timer objects, `wait` blocks the MATLAB command line until all the timers have stopped running.

If the timer is not running, `wait` returns immediately.

See Also

`timer` | `start` | `stop`

Purpose

Open or update wait bar dialog box

Syntax

```
h = waitbar(x,'message')
waitbar(x,'message','CreateCancelBtn','button_callback')
waitbar(x,'message',property_name,property_value,...)
waitbar(x)
waitbar(x,h)
waitbar(x,h,'updated message')
```

Description

A wait bar is a figure that displays what percentage of a calculation is complete as the calculation proceeds by progressively filling a bar with red from left to right.

`h = waitbar(x,'message')` displays a wait bar of fractional length `x`. The wait bar figure displays until the code that controls it closes it or the use clicks its Close Window button. Its (figure) handle is returned in `h`. The argument `x` must be between `0` and `1`.

Note Wait bars are not modal figures (their `WindowStyle` is `'normal'`). They often appear to be modal because the computational loops within which they are called prevent interaction with the Command Window until they terminate. For more information, see `WindowStyle` in the MATLAB Figure Properties documentation.

`waitbar(x,'message','CreateCancelBtn','button_callback')` specifying `CreateCancelBtn` adds a **Cancel** button to the figure that executes the MATLAB commands specified in `button_callback` when the user clicks the **Cancel** button or the **Close Figure** button. `waitbar` sets both the **Cancel** button callback and the figure `CloseRequestFcn` to the string specified in `button_callback`.

`waitbar(x,'message',property_name,property_value,...)` optional arguments `property_name` and `property_value` enable you to set figure properties for the `waitbar`.

`waitbar(x)` subsequent calls to `waitbar(x)` extend the length of the bar to the new position `x`. Successive values of `x` normally increase. If they decrease, the wait bar runs in reverse.

`waitbar(x,h)` extends the length of the bar in the wait bar `h` to the new position `x`.

`waitbar(x,h,'updated message')` updates the message text in the `waitbar` figure, in addition to setting the fractional length to `x`.

Examples

Example 1 — Basic Wait Bar

Typically, you call `waitbar` repeatedly inside a `for` loop that performs a lengthy computation. For example:

```
h = waitbar(0,'Please wait...');
steps = 1000;
for step = 1:steps
    % computations take place here
    waitbar(step / steps)
end
close(h)
```

Example 2 — Wait Bar with Dynamic Text and Cancel Button

Adding a **Cancel** button allows user to abort the computation. Clicking it sets a logical flag in the figure's application data (appdata). The function tests for that value within the main loop and exits the loop as soon as the flag has been set. The example iteratively approximates the value of π. At each step, the current value is encoded as a string and displayed in the wait bar's message field. When the function finishes,

it destroys the wait bar and returns the current estimate of π and the number of steps it ran.

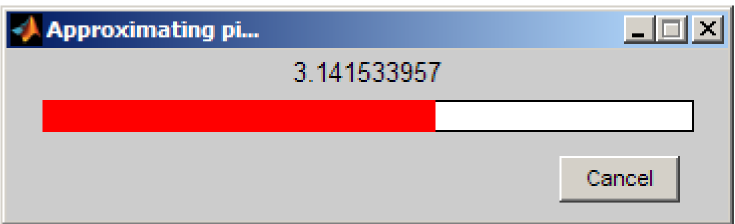
Copy the following function to a code file and save it as `approxpi.m`. Execute it as follows, allowing it to run for 10,000 iterations.

```
[estimated_pi steps] = approxpi(10000)
```

You can click **Cancel** or close the window to abort the computation and return the current estimate of π.

```
function [valueofpi step] = approxpi(steps)
% Converge on pi in steps iterations, displaying waitbar.
% User can click Cancel or close button to exit the loop.
% Ten thousand steps yields error of about 0.001 percent.

h = waitbar(0,'1','Name','Approximating pi...',...
            'CreateCancelBtn',...
            'setappdata(gcbf,''canceling'',1)');
setappdata(h,'canceling',0)
% Approximate as pi^2/8 = 1 + 1/9 + 1/25 + 1/49 + ...
pisqover8 = 1;
denom = 3;
valueofpi = sqrt(8 * pisqover8);
for step = 1:steps
    % Check for Cancel button press
    if getappdata(h,'canceling')
        break
    end
    % Report current estimate in the waitbar's message field
    waitbar(step/steps,h,sprintf('%12.9f',valueofpi))
    % Update the estimate
    pisqover8 = pisqover8 + 1 / (denom * denom);
    denom = denom + 2;
    valueofpi = sqrt(8 * pisqover8);
end
delete(h)       % DELETE the waitbar; don't try to CLOSE it.
```

The function sets the figure `Name` property to describe what is being computed. In the `for` loop, calling `waitbar` sets the fractional progress indicator and displays intermediate results. the code `waitbar(i/steps,h,sprintf('%12.9f',valueofpi))` sets the wait bar's `message` variable to a string representation of the current estimate of *pi*. Naturally, the extra computation involved makes iterations last longer than they need to, but such feedback can be helpful to users.



Note You should call `delete` to remove a wait bar when you give it a `CloseRequestFcn`, as in the preceding code; calling `close` does not close it, and makes its Cancel and Close Window buttons unresponsive. This happens because the figure's `CloseRequestFcn` recursively calls itself. In such a situation you must forcibly remove the wait bar, for example like this:

```
set(0,'ShowHiddenHandles','on')
delete(get(0,'Children'))
```

However, as issuing these commands will delete all open figures—not just the wait bar—it is best never to use `close` in a `CloseRequestFcn` to close a window.

See Also

`close` | `delete` | `dialog` | `msgbox` | `getappdata` | `setappdata`

Purpose

Block execution and wait for event or condition

Syntax

```
waitfor(h)
waitfor(h,'PropertyName')
waitfor(h,'PropertyName',PropertyValue)
```

Description

`waitfor(h)` blocks the caller from executing statements until the graphics object identified by handle `h` closes (is deleted). `h` must be scalar. When object `h` no longer exists, `waitfor` returns, enabling execution to resume. If the object does not exist, `waitfor` returns immediately without processing any events.

`waitfor(h,'PropertyName')` blocks the caller from executing until the value of `'PropertyName'` (any property of the graphics object `h`) changes or `h` closes (is deleted). If `'PropertyName'` is not a valid property for the object, `waitfor` returns immediately without processing any events.

`waitfor(h,'PropertyName',PropertyValue)` blocks the caller from executing until the value of `'PropertyName'` for the graphics object `h` changes to the specific value `PropertyValue` or `h` closes (is deleted). If the value of `'PropertyName'` is already `PropertyValue`, `waitfor` returns immediately without processing any events.

Definitions

- `waitfor` blocks execution so that command-line expressions and subsequent statements in the blocked file do not execute until a specified condition occurs.
- While `waitfor` prevents its caller from continuing, callbacks that respond to various events (for example, pressing a mouse button) can still run, unaffected by `waitfor`.
- `waitfor` also blocks Simulink models from executing, but callbacks do still execute.
- `waitfor` can block nested function calls. For example, a callback invoked while `waitfor` is active can invoke `waitfor`.
- If you type **Ctrl+C** in the Command Window while `waitfor` is blocking execution, the executing program terminates. To avoid

terminating, the program can call `waitfor` within a `try/catch` block that handles the exception that typing **Ctrl+C** generates. To learn more, see the third following example.

Examples

Create a plot and pause execution of the rest of the statements until you close the figure window:

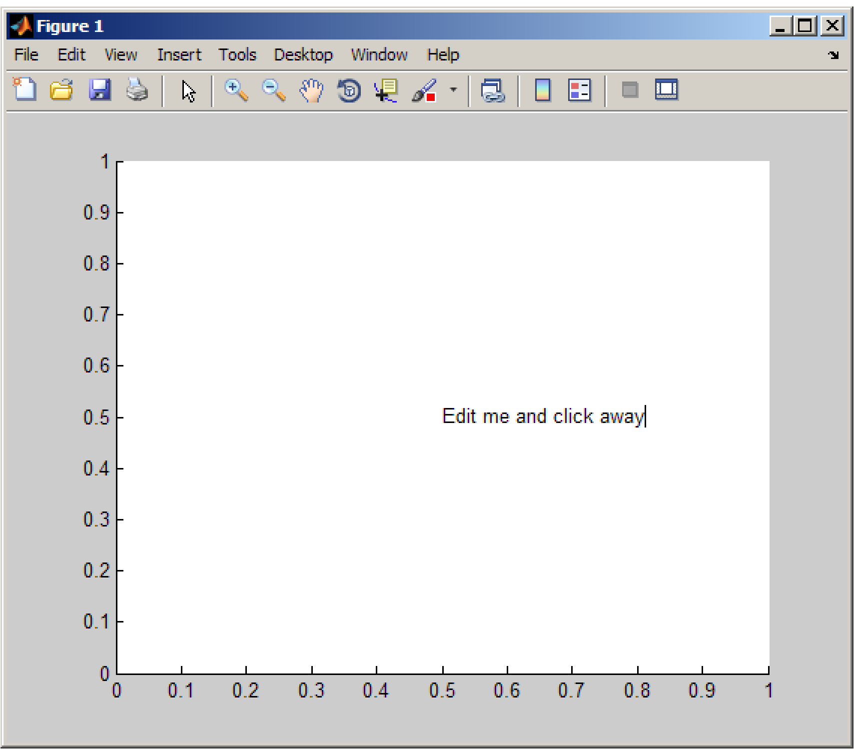
```
f = warndlg('This is a warning.', 'A Warning Dialog');
disp('This prints immediately');
drawnow    % Necessary to print the message
waitfor(f);
disp('This prints after you close the warning dialog');
```

Suspend execution until name of figure changes:

```
f = figure('Name', datestr(now));
h = uicontrol('String', 'Change Name', 'Position', [20 20 100 30], ...
'Callback', 'set(gcbf, ''Name'', datestr(now))');
disp('This prints immediately');
drawnow    % Necessary to print the message
waitfor(f, 'Name');
disp('This prints after button click that changes the figure''s name')
```

Display text object and wait for user to edit it:

```
figure;
textH = text(.5, .5, 'Edit me and click away');
set(textH,'Editing','on', 'BackgroundColor',[1 1 1]);
disp('This prints immediately.');
drawnow
waitfor(textH,'Editing','off');
set(textH,'BackgroundColor',[1 1 0]);
disp('This prints after text editing is complete.');
```



If you close the figure while `waitfor` is executing, an error occurs because the code attempts to access handles of objects that no longer exist. You can handle the error by enclosing code starting with the call to `waitfor` in a try/catch block, as follows:

```
figure;
textH = text(.5, .5, 'Edit me and click away');
set(textH,'Editing','on', 'BackgroundColor',[1 1 1]);
disp('This prints immediately.');
drawnow
% Use try/catch block to handle errors, such as deleting figure
try
    waitfor(textH,'Editing','off');
    set(textH,'BackgroundColor',[1 1 0]);
```

```
    disp('This prints after text editing is complete.');
catch ME
    disp('This prints if figure is deleted:')
    disp(ME.message)
  % You can place other code to respond to the error here
end
```

The ME variable is a MATLAB Exception object that you can use to determine the type of error that occurred. For more information, see "Respond to an Exception".

See Also

`drawnow` | `keyboard` | `pause` | `uiresume` | `uiwait` | `waitforbuttonpress`

How To

- "Control Callback Execution and Interruption"

waitforbuttonpress

Purpose

Wait for key press or mouse-button click

Syntax

`k = waitforbuttonpress`

Description

`k = waitforbuttonpress` blocks the caller's execution stream until the function detects that the user has clicked a mouse button or pressed a key while the figure window is active. The function returns

- `0` if it detects a mouse button click
- `1` if it detects a key press

If a `WindowButtonDownFcn` is defined for the figure, its callback is executed before `waitforbuttonpress` returns a value.

Only keys that generate characters cause the function to return. Pressing any of the following keys by itself does nothing: **Ctrl**, **Shift**, **Alt**, **Caps_lock**, **Num_lock**, **Scroll_lock**.

Additional information about the event that causes execution to resume is available through the figure `CurrentCharacter`, `SelectionType`, and `CurrentPoint` properties.

You can interrupt `waitforbuttonpress` by typing **Ctrl**+C, but an error results unless the function is called from within a `try/catch` block. You also receive an error from `waitforbuttonpress` if you close the figure by clicking the **X** close box unless you call `waitforbuttonpress` within a `try/catch` block.

Examples

These statements display text in the Command Window when the user either clicks a mouse button or types a key in the figure window:

```
w = waitforbuttonpress;
if w == 0
    disp('Button click')
else
    disp('Key press')
end
```

See Also

`dragrect` | `ginput` | `rbbox` | `waitfor`

warndlg

Purpose

Open warning dialog box

Syntax

```
h = warndlg
h = warndlg(warningstring)
h = warndlg(warningstring,dlgname)
h = warndlg(warningstring,dlgname,createmode)
```

Description

`h = warndlg` displays a dialog box named `Warning Dialog` containing the string `This is the default warning string.` The `warndlg` function returns the handle of the dialog box in `h`. The warning dialog box disappears after the user clicks **OK**.

`h = warndlg(warningstring)` displays a dialog box with the title `Warning Dialog` containing the string specified by `warningstring`. The `warningstring` argument can be any valid string format – cell arrays are preferred.

To use multiple lines in your warning, define `warningstring` using either of the following:

- `sprintf` with `newline` characters separating the lines

  ```
  warndlg(sprintf('Message line 1 \n Message line 2'))
  ```

- Cell arrays of strings

  ```
  warndlg({'Message line 1';'Message line 2'})
  ```

`h = warndlg(warningstring,dlgname)` displays a dialog box with title `dlgname`.

`h = warndlg(warningstring,dlgname,createmode)` specifies whether the warning dialog box is modal or nonmodal. Optionally, it can also specify an interpreter for `warningstring` and `dlgname`. The *`createmode`* argument can be a string or a structure.

If *`createmode`* is a string, it must be one of the values shown in the following table.

createmode Value	Description
`modal`	Replaces the warning dialog box having the specified `Title`, that was last created or clicked on, with a modal warning dialog box as specified. All other warning dialog boxes with the same title are deleted. The dialog box which is replaced can be either modal or nonmodal.
`non-modal` (default)	Creates a new nonmodal warning dialog box with the specified parameters. Existing warning dialog boxes with the same title are not deleted.
`replace`	Replaces the warning dialog box having the specified `Title`, that was last created or clicked on, with a nonmodal warning dialog box as specified. All other warning dialog boxes with the same title are deleted. The dialog box which is replaced can be either modal or nonmodal.

Note A modal dialog box prevents the user from interacting with other windows before responding. To block MATLAB program execution as well, use the `uiwait` function.

If you open a dialog with `errordlg`, `msgbox`, or `warndlg` using `'CreateMode','modal'` and a non-modal dialog created with any of these functions is already present and *has the same name as the modal dialog*, the non-modal dialog closes when the modal one opens.

For more information about modal dialog boxes, see `WindowStyle` in the Figure Properties.

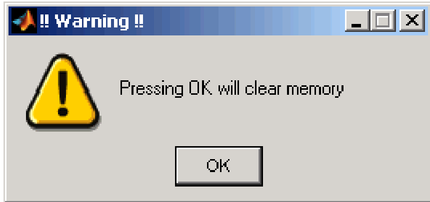
If `CreateMode` is a structure, it can have fields `WindowStyle` and `Interpreter`. `WindowStyle` must be one of the options shown in the table above. `Interpreter` is one of the strings `'tex'` or `'none'`. The default value for `Interpreter` is `'none'`.

Examples

The statement

```
warndlg('Pressing OK will clear memory','!! Warning !!')
```

displays this dialog box:



See Also

`dialog` | `errordlg` | `helpdlg` | `inputdlg` | `listdlg` | `msgbox` | `questdlg` | `figure` | `uiwait` | `uiresume` | `warning`

Purpose

Warning message

Syntax

```
warning('message')
warning('message', a1, a2,...)
warning('message_id', 'message')
warning('message_id', 'message', a1, a2, ..., an)
s = warning(state, 'message_id')
s = warning(state, mode)
```

Description

`warning('message')` displays descriptive text `message` and sets the warning state that `lastwarn` returns. If `message` is an empty string (`''`), `warning` resets the warning state but does not display any text.

`warning('message', a1, a2,...)` displays a message string that contains formatting conversion characters, such as those used with the MATLAB `sprintf` function. Each conversion character in `message` is converted to one of the values `a1`, `a2`, `...` in the argument list.

Note MATLAB converts special characters (like `\n` and `%d`) in the warning message string only when you specify more than one input argument with `warning`. See Example 3 below.

`warning('message_id', 'message')` attaches a unique identifier, or `message_id`, to the warning message. The identifier enables you to single out certain warnings during the execution of your program, controlling what happens when the warnings are encountered.

`warning('message_id', 'message', a1, a2, ..., an)` includes formatting conversion characters in `message`, and the character translations in arguments `a1, a2, ..., an`.

`s = warning(state, 'message_id')` is a warning control statement that enables you to indicate how you want MATLAB to act on certain warnings. The `state` argument can be `'on'`, `'off'`, or `'query'`. The `message_id` argument can be a message identifier string, `'all'`, or `'last'`.

Output `s` is a structure array that indicates the previous state of the selected warnings. The structure has the fields `identifier` and `state`.

`s = warning(state, mode)` is a warning control statement that enables you to display a stack trace or display more information with each warning. The `state` argument can be `'on'`, `'off'`, or `'query'`. The `mode` argument can be `'backtrace'` or `'verbose'`.

Examples

Example 1

Generate a warning that displays a simple string:

```
if ~ischar(p1)
   warning('Input must be a string')
end
```

Example 2

Generate a warning string that is defined at run-time. The first argument defines a message identifier for this warning:

```
warning('MATLAB:paramAmbiguous', ...
        'Ambiguous parameter name, "%s".', param)
```

Example 3

MATLAB converts special characters (like `\n` and `%d`) in the warning message string only when you specify more than one input argument with `warning`. In the single argument case shown below, `\n` is taken to mean `backslash-n`. It is not converted to a newline character:

```
warning('In this case, the newline \n is not converted.')
Warning: In this case, the newline \n is not converted.
```

But, when more than one argument is specified, MATLAB does convert special characters. This is true regardless of whether the additional argument supplies conversion values or is a message identifier:

```
warning('WarnTests:convertTest', ...
      'In this case, the newline \n is converted.')
Warning: In this case, the newline
```

```
is converted.
```

Example 4

Turn on one particular warning, saving the previous state of this one warning in s. Remember that this nonquery syntax performs an implicit query prior to setting the new state:

```
s = warning('on', 'Control:parameterNotSymmetric');
```

After doing some work that includes making changes to the state of some warnings, restore the original state of all warnings:

```
warning(s)
```

See Also

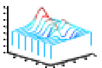
lastwarn | warndlg | error | lasterror | errordlg | dbstop | disp | sprintf

Related Examples

- "Issue Warnings and Errors"
- "Suppress Warnings"
- "Restore Warnings"
- "Change How Warnings Display"

Purpose

Waterfall plot



Syntax

```
waterfall(Z)
waterfall(X,Y,Z)
waterfall(...,C)
waterfall(axes_handles,...)
h = waterfall(...)
```

Description

The `waterfall` function draws a mesh similar to the `meshz` function, but it does not generate lines from the columns of the matrices. This produces a “waterfall” effect.

`waterfall(Z)` creates a waterfall plot using `x = 1:size(Z,2)` and `y = 1:size(Z,1)`. `Z` determines the color, so color is proportional to surface height.

`waterfall(X,Y,Z)` creates a waterfall plot using the values specified in `X`, `Y`, and `Z`. `Z` also determines the color, so color is proportional to the surface height. If `X` and `Y` are vectors, `X` corresponds to the columns of `Z`, and `Y` corresponds to the rows, where `length(x) = n`, `length(y) = m`, and `[m,n] = size(Z)`. `X` and `Y` are vectors or matrices that define the *x*- and *y*-coordinates of the plot. `Z` is a matrix that defines the *z*-coordinates of the plot (i.e., height above a plane). If `C` is omitted, color is proportional to `Z`.

`waterfall(...,C)` uses scaled color values to obtain colors from the current colormap. Color scaling is determined by the range of `C`, which must be the same size as `Z`. MATLAB performs a linear transformation on `C` to obtain colors from the current colormap.

`waterfall(axes_handles,...)` plots into the axes with handle `axes_handle` instead of the current axes (`gca`).

`h = waterfall(...)` returns the handle of the `patch` graphics object used to draw the plot.

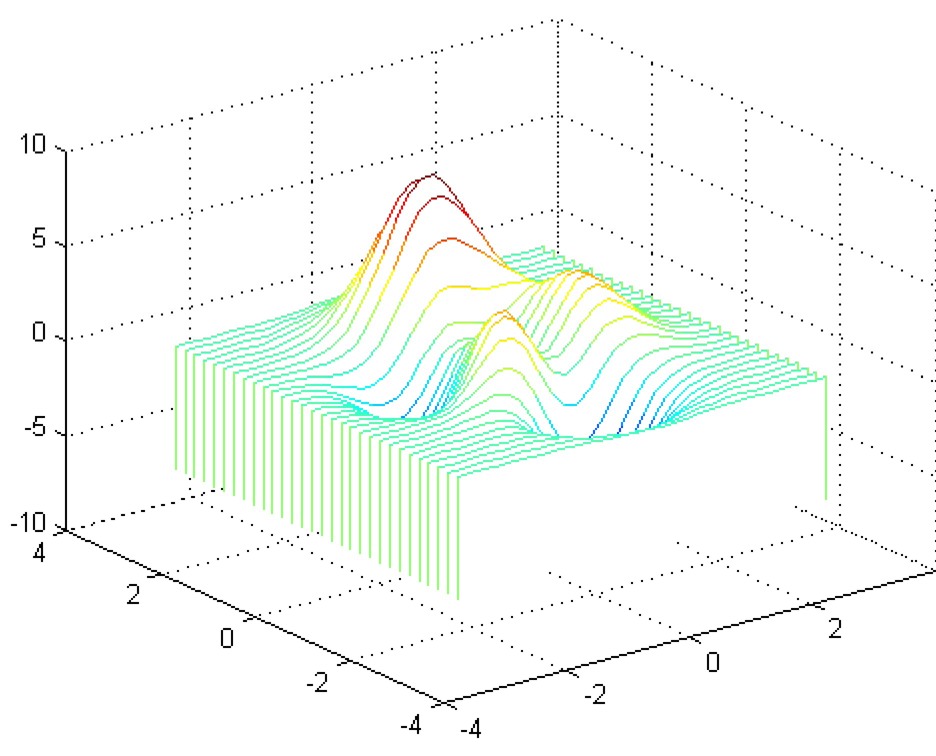
Tips

For column-oriented data analysis, use `waterfall(Z')` or `waterfall(X',Y',Z')`.

Examples

Produce a waterfall plot of the `peaks` function.

```
figure
[X,Y,Z] = peaks(30);
waterfall(X,Y,Z)
```



Algorithms

The range of X, Y, and Z, or the current setting of the axes Xlim, YLim, and ZLim properties, determines the range of the axes (also set by axis). The range of C, or the current setting of the axes CLim property, determines the color scaling (also set by caxis).

The CData property for the patch graphics objects specifies the color at every point along the edge of the patch, which determines the color of the lines.

The waterfall plot looks like a mesh surface; however, it is a patch graphics object. To create a surface plot similar to waterfall, use the meshz function and set the MeshStyle property of the surface to 'Row'. For a discussion of parametric surfaces and related color properties, see surf.

See Also

axes | axis | caxis | meshz | ribbon | surf

Purpose

Information about WAVE (`.wav`) sound file

Note `wavfinfo` will be removed in a future release. Use `audioinfo` instead.

Syntax

`[m d] = wavfinfo(filename)`

Description

`[m d] = wavfinfo(filename)` returns information about the contents of the WAVE sound file specified by the string *`filename`*. Enclose the *`filename`* input in single quotes.

`m` is the string `'Sound (WAV) file'`, if *`filename`* is a WAVE file. Otherwise, it contains an empty string (`''`).

`d` is a string that reports the number of samples in the file and the number of channels of audio data. If *`filename`* is not a WAVE file, it contains the string `'Not a WAVE file'`.

See Also

`audioplayer` | `audiorecorder` | `audioread` | `audiowrite`

wavplay

Purpose

Play recorded sound on PC-based audio output device

Note `wavplay` will be removed in a future release. Use `audioplayer` instead.

Syntax

```
wavplay(y,Fs)
wavplay(y,Fs,mode)
```

Description

`wavplay(y,Fs)` plays the audio signal stored in the vector `y` on a PC-based audio output device. `Fs` is the integer sample rate in Hz (samples per second). The default value for `Fs` is 11025 Hz. `wavplay` supports only 1- or 2-channel (mono or stereo) audio signals. To play in stereo, `y` must be a two-column matrix.

`wavplay(y,Fs,mode)` specifies how `wavplay` interacts with the command line. The string `mode` is one of the following:

- `'sync'` (default): You do not have access to the command line until the sound has finished playing (a blocking device call).
- `'async'`: You have immediate access to the command line as soon as the sound begins to play on the audio output device (a nonblocking device call). If you call `wavplay` again in `async` mode while the audio is playing, `wavplay` blocks access to the command line until the original playback completes.

The audio signal `y` can be one of four data types. The number of bits used to quantize and play back each sample depends on the data type.

Data Types for wavplay

Data Type	Quantization
Double-precision (default value)	16 bits/sample
Single-precision	16 bits/sample

Data Types for wavplay (Continued)

Data Type	Quantization
16-bit signed integer	16 bits/sample
8-bit unsigned integer	8 bits/sample

Tips

The `wavplay` function is for use only with 32-bit Microsoft Windows operating systems. To play audio data on other platforms, use `audioplayer`.

Examples

The MAT-files `gong.mat` and `chirp.mat` both contain an audio signal `y` and a sampling frequency `Fs`. Load and play the gong and the chirp audio signals. Change the names of these signals in between `load` commands and play them sequentially using the `'sync'` option for `wavplay`.

```
load chirp;
y1 = y; Fs1 = Fs;
load gong;
wavplay(y1,Fs1,'sync') % The chirp signal finishes before the
wavplay(y,Fs)          % gong signal begins playing.
```

See Also

`audioplayer` | `audioinfo` | `audioread` | `audiowrite`

wavread

Purpose

Read WAVE (`.wav`) sound file

Note `wavread` will be removed in a future release. Use `audioread` instead.

Syntax

```
y = wavread(filename)
[y, Fs] = wavread(filename)
[y, Fs, nbits] = wavread(filename)
[y, Fs, nbits, opts] = wavread(filename)
[ ___ ] = wavread(filename, N)
[ ___ ] = wavread(filename, [N1 N2])
[ ___ ] = wavread( ___ , fmt)
siz = wavread(filename,'size')
```

Description

`y = wavread(filename)` loads a WAVE file specified by the string *`filename`*, returning the sampled data in *`y`*. If *`filename`* does not include an extension, `wavread` appends `.wav`.

`[y, Fs] = wavread(filename)` returns the sample rate (*`Fs`*) in Hertz used to encode the data in the file.

`[y, Fs, nbits] = wavread(filename)` returns the number of bits per sample (*`nbits`*).

`[y, Fs, nbits, opts] = wavread(filename)` returns a structure *`opts`* of additional information contained in the WAV file. The content of this structure differs from file to file. Typical structure fields include `opts.fmt` (audio format information) and `opts.info` (text that describes the title, author, etc.).

`[ ___ ] = wavread(filename, N)` returns only the first *`N`* samples from each channel in the file.

`[ ___ ] = wavread(filename, [N1 N2])` returns only samples *`N1`* through *`N2`* from each channel in the file.

`[ ___ ] = wavread( ___ , fmt)` specifies the data format of `y` used to represent samples read from the file. `fmt` can be either of the following values, or a partial match (case-insensitive):

`'double'`	Double-precision normalized samples (default).
`'native'`	Samples in the native data type found in the file.

`siz = wavread(filename,'size')` returns the size of the audio data contained in `filename` instead of the actual audio data, returning the vector `siz = [samples channels]`.

Output Scaling

The range of values in `y` depends on the data format `fmt` specified. Some examples of output scaling based on typical bit-widths found in a WAV file are given below for both `'double'` and `'native'` formats.

Native Formats

Number of Bits	MATLAB Data Type	Data Range
8	`uint8` (unsigned integer)	0 <= y <= 255
16	`int16` (signed integer)	-32768 <= y <= +32767
24	`int32` (signed integer)	-2^23 <= y <= 2^23-1
32	`single` (floating point)	-1.0 <= y < +1.0

Double Formats

Number of Bits	MATLAB Data Type	Data Range
N<32	double	-1.0 <= *y* < +1.0
N=32	double	-1.0 <= *y* <= +1.0 Note: Values in *y* might exceed -1.0 or +1.0 for the case of N=32 bit data samples stored in the WAV file.

`wavread` supports multi-channel data, with up to 32 bits per sample.

`wavread` supports Pulse-code Modulation (PCM) data format only.

Examples

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

```
% Create WAV file in current folder.
load handel.mat

hfile = 'handel.wav';
wavwrite(y, Fs, hfile)
clear y Fs

% Read the data back into MATLAB, and listen to audio.
[y, Fs, nbits, readinfo] = wavread(hfile);
sound(y, Fs);

% Pause before next read and playback operation.
duration = numel(y) / Fs;
pause(duration + 2)

% Read and play only the first 2 seconds.
nsamples = 2 * Fs;
[y2, Fs] = wavread(hfile, nsamples);
```

```
sound(y2, Fs);
pause(4)

% Read and play the middle third of the file.
sizeinfo = wavread(hfile, 'size');

tot_samples = sizeinfo(1);
startpos = tot_samples / 3;
endpos = 2 * startpos;

[y3, Fs] = wavread(hfile, [startpos endpos]);
sound(y3, Fs);
```

See Also

mmfileinfo | audioinfo | audiowrite | audioread | audioplayer | audiorecorder | sound

wavrecord

Purpose

Record sound using PC-based audio input device

Note `wavrecord` will be removed in a future release. Use `audiorecorder` instead.

Syntax

```
y = wavrecord(n,Fs)
y = wavrecord(___,ch)
y = wavrecord(___,'dtype')
```

Description

`y = wavrecord(n,Fs)` records `n` samples of an audio signal, sampled at a rate of `Fs` Hz (samples per second). The default value for `Fs` is 11025 Hz.

`y = wavrecord(___,ch)` uses `ch` number of input channels from the audio device. `ch` can be either 1 or 2, for mono or stereo, respectively. The default value for `ch` is 1.

`y = wavrecord(___,'dtype')` uses the data type specified by the string `'dtype'` to record the sound. The following table lists the string values for `'dtype'` along with the corresponding bits per sample and acceptable data range for `y`.

dtype	Bits/sample	y Data Range
`'double'`	16	–1.0 <= y < +1.0
`'single'`	16	–1.0 <= y < +1.0
`'int16'`	16	–32768 <= y <= +32767
`'uint8'`	8	0 <= y <= 255

Tips

Standard sampling rates for PC-based audio hardware are 8000, 11025, 22050, and 44100 samples per second. Stereo signals are returned as two-column matrices. The first column of a stereo audio matrix corresponds to the left input channel, while the second column corresponds to the right input channel.

The wavrecord function is for use only with 32-bit Microsoft Windows operating systems. To record audio data from audio input devices on other platforms, use audiorecorder.

Examples

Record 5 seconds of 16-bit audio sampled at 11025 Hz. Play back the recorded sound using wavplay. Speak into your audio device (or produce your audio signal) while the wavrecord command runs.

```
Fs = 11025;
y = wavrecord(5*Fs,Fs,'int16');
wavplay(y,Fs);
```

See Also

audiorecorder | audioinfo | audioread | audiowrite

wavwrite

Purpose

Write WAVE (`.wav`) sound file

Note `wavwrite` will be removed in a future release. Use `audiowrite` instead.

Syntax

```
wavwrite(y,filename)
wavwrite(y,Fs,filename)
wavwrite(y,Fs,N,filename)
```

Description

`wavwrite(y,filename)` writes the data stored in the variable `y` to a WAVE file called `filename`. The `filename` input is a string enclosed in single quotes. The data has a sample rate of 8000 Hz and is assumed to be 16-bit. Each column of the data represents a separate channel. Therefore, stereo data should be specified as a matrix with two columns.

`wavwrite(y,Fs,filename)` writes the data stored in the variable `y` to a WAVE file called `filename`. The data has a sample rate of `Fs` Hz and is assumed to be 16-bit.

`wavwrite(y,Fs,N,filename)` writes the data stored in the variable `y` to a WAVE file called `filename`. The data has a sample rate of `Fs` Hz and is `N`-bit, where `N` is 8, 16, 24, or 32.

Input Data Ranges

The range of values in `y` depends on the number of bits specified by `N` and the data type of `y`. The following tables list the valid input ranges based on the value of `N` and the data type of `y`.

If `y` contains integer data:

N Bits	y Data Type	y Data Range	Output Format
8	`uint8`	0 <= *y* <= 255	`uint8`
16	`int16`	–32768 <= *y* <= +32767	`int16`
24	`int32`	–2^23 <= *y* <= 2^23 – 1	`int32`

If y contains floating-point data:

N Bits	y Data Type	y Data Range	Output Format
8	single or double	–1.0 <= *y* < +1.0	uint8
16	single or double	–1.0 <= *y* < +1.0	int16
24	single or double	–1.0 <= *y* < +1.0	int32
32	single or double	–1.0 <= *y* <= +1.0	single

For floating point data where *N* < 32, amplitude values are clipped to the range –1.0 <= *y* < +1.0.

Note 8-, 16-, and 24-bit files are type 1 integer pulse code modulation (PCM). 32-bit files are written as type 3 normalized floating point.

See Also

mmfileinfo | audioinfo | audioread | audiowrite | audioplayer | audiorecorder | sound

Purpose

Open Web page or file in browser

Syntax

```
web
web(url)
web(url,opt)
web(url,opt1,...,optN)

stat = web(___)
[stat,h] = web(___)
[stat,h,url] = web(___)
```

Description

`web` opens an empty MATLAB Web browser.

`web(url)` opens the page specified by `url` in the MATLAB Web browser. If multiple browsers are open, the page displays in the one that was most recently used.

`web(url,opt)` opens the page using the specified browser option, such as `'-new'` to create a new browser instance or `'-browser'` to use the system browser.

`web(url,opt1,...,optN)` opens the page using one or more browser options.

`stat = web(___)` returns the status of the operation: `0` if successful, `1` or `2` if unsuccessful. You can include any of the input arguments in previous syntaxes.

`[stat,h] = web(___)` returns a handle to a MATLAB Web browser that allows you to close it using the command `close(h)`. If you do not specify any inputs to the `web` function, such as `[stat,h] = web`, then the handle corresponds to the most recently used MATLAB Web browser.

`[stat,h,url] = web( ___ )` returns the URL of the current page in the MATLAB Web browser.

Input Arguments

url - Web page address or file location

string

Web page address or file location, specified as a string. File locations can include an absolute or relative path.

If `url` corresponds to a file in the installed product documentation, then the page displays in the MATLAB Help browser instead of the Web browser.

Example: `'http:\\www.mathworks.com'`

Example: `'myfolder/myfile.html'`

opt - Browser option

`'-browser'` | `'-new'` | `'-noaddressbox'` | `'-notoolbar'`

Browser option, specified as one of the following strings. Options can appear in any order.

`'-browser'`	Opens the page in a system browser window instead of the MATLAB Web browser. On Microsoft Windows and Apple Macintosh platforms, the operating system determines the system Web browser. On other systems, the default is the Mozilla® Firefox® browser, but you can change the default using MATLAB Web preferences.
`'-new'`	Opens the page in a new MATLAB Web browser window. Does not apply to the system browser.

`'-noaddressbox'`	Opens the page in a browser that does not display the address box. Only applies to new instances of the MATLAB Web browser.
`'-notoolbar'`	Opens the page in a browser that does not display a toolbar or address box. Only applies to new instances of the MATLAB Web browser.

Example: `'-new','-noaddressbox'`

Output Arguments

stat - Browser status

0 | 1 | 2

Browser status, returned as an integer with one of these values:

0	Found and launched system browser.
1	Could not find system browser.
2	Found, but could not launch system browser.

h - Handle to most recent MATLAB Web browser

scalar

Handle to the most recent MATLAB Web browser, returned as a scalar instance of the associated Java class.

If you do not request the handle when you open the page, be aware that this handle might not correspond to your most recent use of the `web` function. Other MATLAB functionality also uses the `web` function, such as links to external sites from the Help browser.

url - Current page address

string

Current page address in the most recent MATLAB Web browser, returned as a string.

Examples

Web Page in MATLAB Web Browser

Open the MathWorks Web site home page.

```
url = 'http://www.mathworks.com';
web(url)
```

Open the page in a new instance of the browser that does not include a toolbar.

```
web(url,'-new','-notoolbar')
```

File in MATLAB Web Browser

View an HTML file that resides on your system.

Create an HTML file by publishing an example program file. Copy the program file to the current folder so that the code can run during the publishing process.

```
program = fullfile(matlabroot,'help','techdoc','matlab_env','examples
copyfile(program);
htmlFile = publish('fourier_demo2.m');
```

View a file by specifying the file name.

```
web(htmlFile)
```

Alternatively, you can use the `file:///` URL scheme, as long as you include the full path. The `publish` function returns the path in the `htmlFile` output.

```
url = ['file:///',htmlFile];
web(url)
```

Web Page in System Browser

Open the MathWorks Web site home page in the system browser.

```
url = 'http://www.mathworks.com';
web(url,'-browser')
```

Email from System Browser

Send email from your system browser's default mail application using the `mailto:` URL scheme.

To run this example, replace the value for `email` with a valid email address.

```
email = 'myaddress@provider.ext';
url = ['mailto:',email];
web(url)
```

Handle to MATLAB Web Browser

Open the MathWorks Web site home page, and then close the browser using its handle.

```
url = 'http://www.mathworks.com';
[stat,h] = web(url);
```

Close the browser window.

```
close(h)
```

Text Displayed in MATLAB Web Browser

View formatted text using the `text://` URL scheme.

```
web('text://<html><h1>Hello World</h1></html>')
```

Tips

- If you plan to deploy an application that calls the `web` function using the MATLAB Compiler product, then use the `'-browser'` option.
- If you are displaying Japanese streaming text in the MATLAB Web browser, specify a header that includes the `charset` attribute. For example:

```
web(['text://<html><head><meta http-equiv="content-type" ' ...

'content="text/html;charset=utf-8"></head><body>TEXT</body></html>'])
```

See Also

`urlread | urlwrite`

Concepts

- "Web Browsers and MATLAB"
- "Specify Proxy Server Settings for Connecting to the Internet"
- "Specify the System Browser for UNIX Platforms"

weekday

Purpose

Day of week

Syntax

```
DayNumber = weekday(D)
[DayNumber,DayName] = weekday(D)
[DayNumber,DayName] = weekday(D,DayForm)
[DayNumber,DayName] = weekday(D,language)
[DayNumber,DayName] = weekday(D,DayForm,language)
```

Description

`DayNumber = weekday(D)` returns a number representing the day of the week for each element in `D`.

`[DayNumber,DayName] = weekday(D)` additionally returns abbreviated English names for the day of the week, in `DayName`.

`[DayNumber,DayName] = weekday(D,DayForm)` returns the name for the day of the week in the format specified by `DayForm`, in US English.

`[DayNumber,DayName] = weekday(D,language)` returns the abbreviated name for the day of the week in the language of the locale specified in `language`.

`[DayNumber,DayName] = weekday(D,DayForm,language)` returns the name for the day of the week in the specified format and in the language of the specified locale. You can specify `DayForm` and `language` in either order.

Input Arguments

D - Serial date numbers or date strings

vector | matrix | string | cell array of strings | character array

Serial date numbers or date strings. Date numbers can be specified as a vector or matrix. Date strings can be specified as a single string, a cell array of strings, or a character array. If `D` is a cell array of strings, it must be 1-by-`n` or `n`-by-1.

If `D` is a single date string, a cell array of date strings, or a character array of date strings, the date strings can be in one of the following formats.

Date String Format	Example
dd-mmm-yyyy	01-Mar-2000
mm/dd/yyyy	03/01/2000
yyyy-mm-dd	2000-03-01

For date strings in other formats, first convert them to serial date numbers using the `datenum` function, before passing them to `weekday`.

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

DayForm - Format of output day names

'short' (default) | 'long'

Format of the output day names, specified as one of the following strings.

DayForm	Format of DayName Names	Example
'short'	Abbreviated name	Mon
'long'	Full name	Monday

language - Output language of day names

'en_US' (default) | 'local'

Output language of day names in `DayName`, specified as one of the following strings.

language	Description
'en_US'	US English
'local'	Language of the current locale

Output Arguments

DayNumber - Value representing day of week

array of integers in the range [1,7]

Value representing the day of the week, returned as an array of integers in the range [1,7], where 1 represents Sunday, and 7 represents Saturday.

- If input D is a numeric array, then the size of DayNumber is equivalent to the size of D.
- If input D is a cell array of strings, then DayNumber is an m-by-1 vector, where m is equivalent to the length of D.

DayName - Name of day of week

character array

Name of the day of the week, returned as a character array. The content of DayName depends on DayForm.

- If DayForm is 'short', then DayName contains an abbreviated name (for example, Tues).
- If DayForm is 'long', then DayName contains the full name of the weekday (for example, Tuesday).

DayName is m-by-n, where m is the number of serial date numbers or date strings in D.

Examples

Return Day of Week of a Date String

Determine the day of the week of December 21, 2012.

```
D = '21-Dec-2012';
[DayNumber,DayName] = weekday(D)
```

```
DayNumber =

     6


DayName =

Fri
```

December 21, 2012 falls on a Friday.

Return Full Day Names of Multiple Date Numbers

Return the full name of the day of the week for a vector of serial date numbers.

```
D = [734999;735015];
DayForm = 'long';
[DayNumber,DayName] = weekday(D,DayForm)

DayNumber =

     5
     7


DayName =

Thursday
Saturday
```

Return Full Day Names in Local Language

Return day names in U.S. English using the `language` input argument.

```
D = 728647;
DayForm = 'long';
language = 'en_US';
```

```
[DayNumber, DayName] = weekday(D,DayForm,language)

DayNumber =

     2


DayName =

Monday
```

In U.S. English, the name of the day of the week is `Monday`.

Return day names in the language of the current locale.

```
language = 'local';
[DayNumber, DayName] = weekday(D,DayForm,language)

DayNumber =

     2


DayName =

Lundi
```

In a French locale, the name of the day of the week is `Lundi`.

Return Day of Week of a Date String in Custom Format

Determine the day of the week for a date specified in the format `mmm.dd.yyyy`. Call `datenum` inside of `weekday` to specify the format of the input date string.

```
[DayNumber,DayName] = weekday(datenum('Dec.21.2012','mmmm.dd.yyyy'))

DayNumber =
```

```
     6


DayName =

Fri
```

See Also

datenum | datevec | eomday

what

Purpose

List MATLAB files in folder

Syntax

```
what
what folderName
what className
what packageName
s = what('folderName')
```

Description

`what` lists the path for the current folder, and lists all files and folders relevant to MATLAB found in the current folder. Files listed are M, MAT, MEX, MDL, and P-files. Folders listed are all class and package folders.

`what` *`folderName`* lists path, file, and folder information for *`folderName`*.

`what` *`className`* lists path, file, and folder information for method folder `@`*`className`*. For example, `what cfit` lists the MATLAB files and folders in `toolbox/curvefit/curvefit/@cfit`.

`what` *`packageName`* lists path, file, and folder information for package folder `+`*`packageName`*. For example, `what commsrc` lists the MATLAB files and folders in `toolbox/comm/comm/+commsrc`.

`s = what('`*`folderName`*`')` returns the results in a structure array with the fields shown in the following table.

Field	Description
`path`	Path to folder
`m`	Cell array of MATLAB program file names
`mat`	Cell array of MAT-file names
`mex`	Cell array of MEX-file names
`mdl`	Cell array of MDL-file names
`slx`	Cell array of SLX-file names
`p`	Cell array of P-file names

Field	Description
`classes`	Cell array of class folders
`packages`	Cell array of package folders

Examples

List the MATLAB files and folders in `C:\Program Files\MATLAB\Rnnnn\toolbox\matlab\audiovideo\+audiovideo`, where *Rnnnn* represents the folder for the MATLAB release, for example, R2012b:

```
what audiovideo

MATLAB Code files in folder C:\Program Files\MATLAB\Rnnnn\toolbox\matlab\audiovideo\+audiovideo

FileFormatInfo

Packages in folder C:\Program Files\MATLAB\Rnnnn\toolbox\matlab\audiovideo\+audiovideo

internal  writer


MATLAB Code files in folder C:\Program Files\MATLAB\Rnnnn\toolbox\matlab\audiovideo

Contents            aufinfo             mmcompinfo          wavplay
audiodevinfo        auread              mmfileinfo          wavread
audioinfo           auwrite             movie2avi           wavrecord
audioplayerreg      avgate              mu2lin              wavwrite
audioread           avifinfo            prefspanel
audiorecorderreg    aviinfo             sound
audiouniquename     aviread             soundsc
audiowrite          lin2mu              wavfinfo

MAT-files in folder C:\Program Files\MATLAB\Rnnnn\toolbox\matlab\audiovideo

chirp          handel          splat
gong           laughter        train
```

```
Classes in folder C:\Program Files\MATLAB\Rnnnn\toolbox\matlab\audiovideo

VideoReader    audioplayer    avifile
VideoWriter    audiorecorder  mmreader

Packages in folder C:\Program Files\MATLAB\Rnnnn\toolbox\matlab\audiovideo

audiovideo
```

Obtain a structure array containing the file and folder names in `toolbox/matlab/codetools` that are relevant to MATLAB, where *Rnnnn* represents the folder for the MATLAB release, for example, R2012a:

```
s = what('codetools')

s =

        path: 'C:\Program Files\MATLAB\Rnnnn\toolbox\matlab\codetools'
           m: {76x1 cell}
         mat: {0x1 cell}
         mex: {0x1 cell}
         mdl: {0x1 cell}
           p: {0x1 cell}
     classes: {2x1 cell}
    packages: {3x1 cell}
```

Find the supporting files for one of the packages in the Communications System Toolbox product:

```
p1 = what('comm');
p1.packages
ans =
```

```
    'gpu'
    'internal'


ans =

    'internal'.
.
.
p2 = what('commsrc');
p2.m
ans =
    'abstractJitter.m'
    'abstractPulse.m'
    'combinedjitter.m'
    'diracjitter.m'
    'periodicjitter.m'
    'randomjitter.m'
```

Alternatives

Use the Current Folder browser to view the list of files in a folder.

See Also

`dir` | `exist` | `lookfor` | `ls` | `which` | `who`

How To

- "Working with Files and Folders"

whatsnew

Purpose Release Notes

Note `whatsnew` will be removed in a future release.

Syntax `whatsnew`

Description `whatsnew` displays the MATLAB Release Notes in the Help browser, presenting information about new features, problems from previous releases that have been fixed in the current release, and compatibility issues.

See Also `help` | `version`

Purpose

Locate functions and files

Syntax

```
which item
which fun1 in fun2

which ___ -all

str = which(item)
str = which(fun1,'in',fun2)

str = which(___,'-all')
```

Description

`which item` displays the full path for `item`.

- If `item` is a MATLAB function in an M or P file, or a Simulink model in an MDL file, then `which` displays the full path for the corresponding file. `item` must be on the MATLAB path.
- If `item` is a method in a loaded Java class, then `which` displays the package, class, and method name for that method.
- If `item` is a workspace variable, then `which` displays a message identifying `item` as a variable.
- If `item` is a file name including the extension, and it is in the current working folder or on the MATLAB path. then `which` displays the full path of `item`.

If `item` is an overloaded function or method, then `which item` returns only the path of the first function or method found.

`which fun1 in fun2` displays the path to function `fun1` that is called by file `fun2`. Use this syntax to determine whether a local function is being called instead of a function on the path. This syntax does not locate nested functions.

`which ___ -all` displays the paths to all items on the MATLAB path with the requested name. Such items include methods of instantiated

classes. You can use `-all` with the input arguments of any of the previous syntaxes.

`str = which(item)` returns the full path for `item` in the string, `str`.

`str = which(fun1,'in',fun2)` returns the path to function `fun1` that is called by file `fun2`. Use this syntax to determine whether a local function is being called instead of a function on the path. This syntax does not locate nested functions.

`str = which(___,'-all')` returns the results of `which` in the string or cell array of strings, `str`. You can use this syntax with any of the input arguments in the previous syntax group.

Input Arguments

item - Function or file to locate

string

Function or file to locate, specified as a string. When using the function form of `which`, enclose all input strings in single quotes. `item` can be in one of the following forms.

Form of the `item` Input	Path to Display
`fun`	Display full path for `fun`, which can be a MATLAB function, Simulink model,
`classname``/`*`method`*	Display full path for the file that defines a method in a MATLAB class. For
`private/`*`fun`*	Limit the search to private functions named `fun`. For example, `which private/orthog` or `which('private/orthog')` displays the path for `orthog.m` in the `/private` subfolder of the parent folder.
`classname``/private/`*`function`*	Limit the search to private methods defined by the MATLAB class, `classname`.

Form of the `item` Input	Path to Display
`fun1 in fun2`	Display the path to function `fun1` that is called by file `fun2`. Use this syntax to determine whether a local function is being called instead of a function on the path. This syntax does not locate nested functions or variables.
`fun(a1,...,an)`	Display the path to the implementation of function `fun` which would be invoked if called with the input arguments `a1,...,an`. Use this syntax to query overloaded functions. See the Example, "Locate Function Invoked with Given Input Arguments" on page 1-6131.

fun1 - Function to locate

string

Function to locate, specified as a string. When using the function form of `which`, enclose all input strings in single quotes, for example, `'myfun1'`.

fun2 - Calling file

string

Calling file, specified as a string. When using the function form of `which`, enclose all input strings in single quotes, for example, `'myfile'`.

Output Arguments

str - Function or file location

string | cell array of strings

Function or file location, returned as a string, or returned as a cell array of strings if you use `'-all'`.

- If `item` is a workspace variable, then `str` is the string `'variable'`.
- If `str` is a cell array of strings, then each row of `str` identifies a function, and the functions are in order of precedence.

Examples

Locate MATLAB Function

Locate the `pinv` function.

```
which pinv
```

```
matlabroot\toolbox\matlab\matfun\pinv.m
```

`pinv` is in the `matfun` folder of MATLAB.

You also can use function syntax to return the path to a string, `str`. When using the function form of `which`, enclose all input strings in single quotes.

```
str = which('pinv');
```

Locate Method in a Loaded Java Class

Create an instance of the Java class. This loads the class into MATLAB.

```
myDate = java.util.Date;
```

Locate the `setMonth` method.

```
which setMonth
```

```
setMonth is a Java method  % java.util.Date method
```

Locate Function in a MATLAB Class

Find the `fopen` function used on MATLAB `serial` class objects.

```
which serial/fopen
```

```
matlabroot\toolbox\matlab\iofun\@serial\fopen.m
% serial method
```

MATLAB displays the path for `fopen.m` in the class folder, `@serial`.

Locate Private Function

Find the `orthog` function in a private folder.

```
which private/orthog
```

```
matlabroot\toolbox\matlab\elmat\private\orthog.m
% Private to elmat
```

MATLAB displays the path for `orthog.m` in the `/private` subfolder of `toolbox/matlab/elmat`.

Determine if Local Function is Called

Determine which `parseargs` function is called by `area.m`.

```
which parseargs in area
```

```
matlabroot\toolbox\matlab\specgraph\area.m (parseargs)
% Subfunction of area
```

You also can use function syntax to return the path to a string, `str`. When using the function form of `which`, enclose all input strings in single quotes.

```
str = which('parseargs','in','area');
```

Locate Function Invoked with Given Input Arguments

Suppose you have a `matlab.io.MatFile` object that corresponds to the example MAT-file `'topography.mat'`:

```
matObj = matfile('topography.mat');
```

Display the path of the implementation of `who` that is invoked when called with the input argument `(matObj)`.

```
which who(matObj)
```

```
matlabroot\toolbox\matlab\iofun\+matlab\+io\MatFile.m
% matlab.io.MatFile method
```

Store the result to a string.

```
str = which('who(matObj)')
```

```
str =
matlabroot\toolbox\matlab\iofun\+matlab\+io\MatFile.m
```

If you do not specify the input argument `(matObj)`, then `which` returns only the path of the first function or method found.

```
which who
```

```
built-in (matlabroot\\toolbox\matlab\general\who)
```

Locate All Items with Given Name

Display the paths to all items on the MATLAB path with the name `fopen`.

```
which fopen -all
```

```
built-in (matlabroot\toolbox\matlab\iofun\fopen)
matlabroot\toolbox\matlab\iofun\@serial\fopen.m           % serial method
matlabroot\toolbox\shared\instrument\@icinterface\fopen.m  % icinterface m
matlabroot\toolbox\instrument\instrument\@i2c\fopen.m
% i2c method
```

Return Path Names as Strings

Return the results of `which` in the string `str`.

You must use the function form of `which`, enclosing all arguments in parentheses and single quotes.

```
str = which('private/stradd','-all');
whos str
```

```
  Name      Size            Bytes  Class    Attributes

  str       2x1               650  cell
```

Tips

- For more information about how MATLAB uses scope and precedence when calling a function, see “Function Precedence Order”.

Limitations

- When the class is not loaded, `which` only finds methods if they are defined in separate files in an @-folder and are not in any packages.

See Also

`dir` | `doc` | `exist` | `lookfor` | `mfilename` | `path` | `type` | `what` | `who` | `fileparts`

while

Purpose

Repeatedly execute statements while condition is true

Syntax

```
while expression
   statements
end
```

Description

`while` *`expression`*, *`statements`*, `end` repeatedly executes one or more MATLAB program *`statements`* in a loop as long as an expression remains true.

An evaluated expression is true when the result is nonempty and contains all nonzero elements (logical or real numeric). Otherwise, the expression is false.

Expressions can include relational operators (such as `<` or `==`) and logical operators (such as `&&`, `||`, or `~`). MATLAB evaluates compound expressions from left to right, adhering to operator precedence rules.

Tips

- If you inadvertently create an infinite loop (that is, a loop that never ends on its own), stop execution of the loop by pressing **Ctrl+C**.
- To programmatically exit the loop, use a `break` statement. To skip the rest of the instructions in the loop and begin the next iteration, use a `continue` statement.
- You can nest any number of `while` statements. Each `while` statement requires an `end` keyword.
- Within an `if` or `while` expression, all logical operators, including `|` and `&`, short-circuit. That is, if the first part of the expression determines a true or false result, MATLAB does not evaluate the second part of the expression.

Examples

Find the first integer `n` for which `factorial(n)` is a 100-digit number:

```
n = 1;
nFactorial = 1;
while nFactorial < 1e100
    n = n + 1;
```

```
    nFactorial = nFactorial * n;
end
```

Count the number of lines of code in the file `magic.m`, skipping all blank lines and comments:

```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) || strncmp(line,'%',1) || ~ischar(line)
        continue
    end
    count = count + 1;
end
fprintf('%d lines\n',count);
fclose(fid);
```

Find the root of the polynomial $x^3 - 2x - 5$ using interval bisection:

```
a = 0; fa = -Inf;
b = 3; fb = Inf;
while b-a > eps*b
   x = (a+b)/2;
   fx = x^3-2*x-5;
   if fx == 0
      break
   elseif sign(fx) == sign(fa)
      a = x; fa = fx;
   else
      b = x; fb = fx;
   end
end
disp(x)
```

Take advantage of short-circuiting to avoid error or warning messages:

```
x = 42;
while exist('myfunction.m') && (myfunction(x) >= pi)
   disp('Condition is true')
   break
end
```

See Also

`end` | `for` | `break` | `continue` | `return` | `if` | `switch`

Tutorials

- "Relational Operators"
- "Logical Operators"

Purpose

Change axes background color

Syntax

```
whitebg
whitebg(fig)
whitebg(ColorSpec)
whitebg(fig, ColorSpec)
whitebg(fig, ColorSpec)
whitebg(fig)
```

Description

`whitebg` complements the colors in the current figure.

`whitebg(fig)` complements colors in all figures specified in the vector `fig`.

`whitebg(ColorSpec)` and `whitebg(fig, ColorSpec)` change the color of the axes, which are children of the figure, to the color specified by `ColorSpec`. Without a figure specification, `whitebg` or `whitebg(ColorSpec)` affects the current figure and the root's default properties so subsequent plots and new figures use the new colors.

`whitebg(fig, ColorSpec)` sets the default axes background color of the figures in the vector `fig` to the color specified by `ColorSpec`. Other axes properties and the figure background color can change as well so that graphs maintain adequate contrast. `ColorSpec` can be a 1-by-3 RGB color or a color string such as `'white'` or `'w'`.

`whitebg(fig)` complements the colors of the objects in the specified figures. This syntax is typically used to toggle between black and white axes background colors, and is where `whitebg` gets its name. Include the root window handle `(0)` in `fig` to affect the default properties for new windows or for `clf reset`.

Tips

`whitebg` works best in cases where all the axes in the figure have the same background color.

`whitebg` changes the colors of the figure's children, with the exception of shaded surfaces. This ensures that all objects are visible against the new background color. `whitebg` sets the default properties on the root such that all subsequent figures use the new background color.

Examples

Set the background color to blue-gray.

```
whitebg([0 .5 .6])
```

Set the background color to blue.

```
whitebg('blue')
```

See Also

ColorSpec | colordef

Purpose

List variables in workspace

Syntax

```
who
who(variables)
who(location)
who(variables,location)
c = who(variables,location)
```

Description

`who` lists in alphabetical order all variables in the currently active workspace.

`who(variables)` lists only the specified variables.

`who(location)` lists variables in the specified location: `'global'` for the global workspace, or `'-file'` for a MAT-file. For MAT-files, you must also include the file name as an input.

`who(variables,location)` lists the specified variables in the specified location. The `location` input can appear before or after `variables`.

`c = who(variables,location)` stores the names of the variables in cell array `c`. Specifying `variables` and `location` is optional.

Tips

- The `who` function displays the variable list unless you specify an output argument.
- When used within a nested function, the `who` function lists the variables in the workspaces of that function and all functions containing that function, grouped by workspace. This applies whether you call `who` from your function code or from the MATLAB debugger.

Input Arguments

variables

Strings that specify the variables to list. Use one of these forms:

var1, var2, ...	List the specified variables. Use the `'*'` wildcard to match patterns. For example, `who('A*')` lists all variables that start with `A`.
`'-regexp'`, *expressions*	List variables whose names match the specified regular expressions.

Default: `'*'` (all variables)

location

String that indicates whether to list variables from the global workspace or from a file:

`'global'`	Global workspace.
`'-file'`, *filename*	MAT-file. The *filename* input can include the full, relative, or partial path.

Default: `''` (current workspace)

Output Arguments

c

Cell array of strings that correspond to each variable name.

Examples

Display information about variables in the current workspace whose names start with the letter `a`:

```
who a*
```

Show variables stored in MAT-file `durer.mat`:

```
who -file durer
```

This code returns:

```
Your variables are:

X        caption  map
```

Store the variable names from `durer.mat` in cell array `durerInfo`:

```
durerInfo = who('-file', 'durer');
```

Display the contents of cell array `durerInfo`:

```
for k=1:length(durerInfo)
   disp(durerInfo{k})
end
```

This code returns:

```
  X
  caption
  map
```

Suppose that a file `mydata.mat` contains variables with names that start with `java` and end with `Array`. Display information about those variables:

```
 whos -file mydata -regexp \<java.*Array\>
```

Call `who` within a nested function (`get_date`):

```
function who_demo
date_time = datestr(now);

[str pos] = textscan(date_time, '%s%s%s', ...
                     1, 'delimiter', '- :');
```

```
get_date(str);

   function get_date(d)
      day = d{1};
      mon = d{2};
      year = d{3};
      who
   end

end
```

When you run `who_demo`, the `who` function displays the variables by function workspace (although the name of the function does not appear in the output):

```
Your variables are:

d          mon        ans        pos
day        year       date_time  str
```

Alternatives

To view the variables in the workspace, use the Workspace browser. To view the contents of MAT-files, use the Details Panel of the Current Folder Browser.

See Also

`assignin` | `clear` | `dir` | `evalin` | `exist` | `inmem` | `load` | `save` | `what` | `whos` | `workspace`

Purpose

Names of variables in MAT-file

Syntax

```
varlist = who(matObj)
varlist = who(matObj,variables)
```

Description

`varlist = who(matObj)` lists alphabetically all variables in the MAT-file associated with `matObj`. Optionally, returns the list in cell array `varlist`.

`varlist = who(matObj,variables)` lists the specified variables.

Input Arguments

matObj

Object created by the `matfile` function.

variables

Names of variables in the MAT-file corresponding to `matObj`. Use one of these forms:

`var1,...,varN`	Comma-separated list of variable name strings. Optionally, match patterns with the `'*'` wildcard, such as `who(matobj,'A*')`.
`'-regexp',expressions`	Regular expression strings that describe variable names.

Output Arguments

varlist

Cell array of strings that correspond to each variable name.

Examples

Display a list of variables in the example file `topography.mat`:

```
matObj = matfile('topography.mat');
who(matObj)
```

This code returns:

```
Your variables are:

topo        topolegend  topomap1    topomap2
```

See Also matfile | whos

Purpose

List variables in workspace, with sizes and types

Syntax

```
whos
whos(variables)
whos(location)
whos(variables,location)

S = whos( ___ )
```

Description

`whos` displays in alphabetical order all variables in the currently active workspace, with information about their sizes and types.

`whos(variables)` displays only the specified variables.

`whos(location)` displays variables in the specified location.

`whos(variables,location)` displays the specified variables in the specified location. The `location` and `variables` inputs can appear in either order.

`S = whos( ___ )` stores information about the variables in structure array, `S`, using no input arguments, or any of the input arguments from the previous syntaxes.

Input Arguments

variables - Variables to display

strings

Variables to display, specified as one or more strings in one of the following forms.

Form of `Variables` Input	Variable Names
var1 ... varN	List the named variables, specified as individual strings. Use the `'*'` wildcard to match patterns. For example, `whos('A*')` lists all variables in the workspace that start with `A`.
`'-regexp'`,*expr1 ... exprN*	List only the variables that match the regular expressions, specified as strings. For example, `whos('-regexp','^Mon','^Tues')` lists only the variables in the workspace that begin with `Mon` or `Tues`.

location - Location of variables
`'global'` \| `'file'`,*filename*

Location of variables, specified as one of the following strings.

Value of `location`	Location of Variables
`'global'`	Global workspace.
`'-file'`,*filename*	MAT-file. The *filename* input can include the full, relative, or partial path. For example, `whos('file','myFile.mat')` lists all variables in the MAT-file named `myFile.mat`. `whos('-file',`*filename*`)` does not return the sizes of any MATLAB objects in file *filename*.

Output Arguments

S - Information about variables

nested structure array

Information about variables, returned as a nested structure array that contains a scalar `struct` for each variable. Each scalar `struct` contains these fields.

Field	Description
`name`	Name of the variable.
`size`	Dimensions of the variable array.
`bytes`	Number of bytes allocated for the variable array.
`class`	Class of the variable. If the variable has no value, class is `'(unassigned)'`.
`global`	True if the variable is global; otherwise, false.
`sparse`	True if the variable is sparse; otherwise, false.
`complex`	True if the variable is complex; otherwise, false.
`nesting`	Structure with these fields: • `function` — Name of the nested or outer function that defines the variable. • `level` — Nesting level of that function.
`persistent`	True if the variable is persistent; otherwise, false.

Examples

Display Information About Workspace Variables

Display information about variables in the current workspace whose names start with the letter `a`.

```
whos('a*')
```

Display Variables Stored in a MAT-File

Display all variables stored in the MAT-file, `durer.mat`.

```
whos('-file','durer.mat')
```

```
  Name             Size                Bytes  Class     Attributes

  X              648x509             2638656  double
  caption          2x28                  112  char
  map            128x3                  3072  double
```

Show only variables in the MAT-file with names that end with `ion`.

```
whos('-file','durer.mat','-regexp','ion$')
```

```
  Name           Size              Bytes  Class    Attributes

  caption        2x28                112  char
```

Store Variable Information in a Structure Array

Store information about the variables in `durer.mat` in structure array `S`.

```
S = whos('-file','durer.mat');
```

Display the contents of `S`.

```
for k = 1:length(S)
   disp(['  ' S(k).name ...
         '  ' mat2str(S(k).size) ...
         '  ' S(k).class]);
end
```

```
  X  [648 509]  double
  caption  [2 28]  char
  map  [128 3]  double
```

Create Function to Display Variable Attribute Information

In the Editor, create a function that creates variables with various persistent, global, sparse, and complex attributes, and then displays information about them.

```
function show_attributes
persistent p;
global g;
p = 1;
g = 2;
s = sparse(eye(5));
c = [4+5i 9-3i 7+6i];
whos
```

When you call the function, `show_attributes` displays the attributes.

```
show_attributes
```

```
Name      Size            Bytes  Class     Attributes

c         1x3                48  double    complex
g         1x1                 8  double    global
p         1x1                 8  double    persistent
s         5x5               128  double    sparse
```

Call `whos` Within a Nested or Anonymous Function

Create a function, `whos_demo`, that contains a nested function, `get_date`. Call `whos` within the nested function.

```
function whos_demo
date_time = datestr(now);

C = strsplit(date_time,{'-',''});
get_date(C);

   function get_date(d)
      day = d{1};
      mon = d{2};
      year = d{3};
      whos
   end
```

```
end
```

When you run the `whos_demo` function, `whos` displays the variables of the nested `get_date` function, and all functions that contain it, grouped by function workspace. This applies whether you call `whos` from your function code or from the MATLAB debugger.

```
whos_demo

  Name              Size              Bytes  Class           Attributes

  ---- get_date ---------------------------------------------------------
  d                 1x3                 372  cell
  day               1x2                   4  char
  mon               1x3                   6  char
  year              1x13                 26  char

  ---- whos_demo --------------------------------------------------------
  C                 1x3                 372  cell
  ans               0x0                   0  (unassigned)
  date_time         1x20                 40  char
```

When called within an anonymous function, variables in the anonymous function also display in a group, titled with the function's signature

Tips

- You also can view the contents of MAT-files using the Details Panel of the Current Folder browser.

Algorithms

`whos` returns the number of bytes each variable occupies in the workspace, not in a MAT-file. Version 7 MAT-files and later are compressed, so the number of bytes in the file is typically fewer than the number of bytes required in the workspace.

See Also

`clear` | `exist` | `what` | `who`

Concepts

- "What Is the MATLAB Workspace?"

Purpose

Names, sizes, and types of variables in MAT-file

Syntax

```
details = whos(matObj)
details = whos(matObj,variables)
```

Description

`details = whos(matObj)` returns information about all variables in the MAT-file associated with `matObj`.

`details = whos(matObj,variables)` returns information about the specified variables.

Input Arguments

matObj

Object created by the `matfile` function.

variables

Names of variables in the MAT-file corresponding to `matObj`. Use one of these forms:

`var1,...,varN`	Comma-separated list of variable name strings. Optionally, match patterns with the `'*'` wildcard, such as `whos(matobj,'A*')`.
`'-regexp',expressions`	Regular expression strings that describe variable names.

Output Arguments

details

Structure array with these fields (identical to the structure returned by the `whos` function):

`name`	Variable name
`size`	Dimensions of the variable

`bytes`	Number of bytes allocated for the array when you load the entire variable
`class`	Class (data type) of the variable
`global`	Whether the variable is global (`true` or `false`)
`sparse`	Whether the variable is sparse
`complex`	Whether the variable is complex
`nesting`	Structure with these fields: • `function` — Name of the nested or outer function that defines the variable • `level` — Nesting level
`persistent`	Whether the variable is persistent

Examples

Display a list of variables in the example file `topography.mat`:

```
matObj = matfile('topography.mat');
whos(matObj)
```

This code returns:

```
  Name              Size                Bytes  Class     Attributes

  topo            180x360              518400  double
  topolegend        1x3                    24  double
  topomap1         64x3                  1536  double
  topomap2        128x3                  3072  double
```

Without loading any data, find the size and number of dimensions of the variable `topo` in `topography.mat`:

```
matObj = matfile('topography.mat');
info = whos(matObj,'topo');
```

```
sizeX = info.size
nDimsX = length(sizeX)
```

This code returns:

```
sizeX =
   180   360

nDimsX =
     2
```

See Also

`matfile` | `size`

wilkinson

Purpose

Wilkinson's eigenvalue test matrix

Syntax

```
W = wilkinson(n)
```

Description

`W = wilkinson(n)` returns one of J. H. Wilkinson's eigenvalue test matrices. It is a symmetric, tridiagonal matrix with pairs of nearly, but not exactly, equal eigenvalues.

Examples

```
wilkinson(7)

ans =

     3     1     0     0     0     0     0
     1     2     1     0     0     0     0
     0     1     1     1     0     0     0
     0     0     1     0     1     0     0
     0     0     0     1     1     1     0
     0     0     0     0     1     2     1
     0     0     0     0     0     1     3
```

The most frequently used case is `wilkinson(21)`. Its two largest eigenvalues are both about 10.746; they agree to 14, but not to 15, decimal places.

See Also

`eig` | `gallery` | `pascal`

Purpose

Open file in appropriate application (Windows)

Syntax

```
winopen(fileName)
```

Description

`winopen(fileName)` opens `fileName` in the associated Microsoft Windows application. The application is associated with the extension in `fileName` in the Windows operating system. `filename` is a string enclosed in single quotes. `winopen` uses a Windows shell command, and performs the same action as double-clicking the file in the Windows Explorer program. That is, `winopen` calls the application associated the file extension to open the file. Use an absolute or relative path for `fileName`.

Examples

Open the file `thesis.doc`, located in the current folder, in the Microsoft Word program:

```
winopen('thesis.doc')
```

Open `myresults.html` in the system Web browser:

```
winopen('D:/myfiles/myresults.html')
```

On Microsoft Windows platforms, open the current folder in the Windows Explorer tool:

```
winopen(cd)
```

To open a file on the MATLAB path, use `winopen` with `which`. For example, to open the `meshgrid` function in the Editor, use:

```
winopen(which('meshgrid'))
```

See Also

`dos` | `open` | `web`

How To

- "Opening and Running Files"

Purpose

Item from Windows registry

Syntax

```
valnames = winqueryreg('name', 'rootkey', 'subkey')
value = winqueryreg('rootkey', 'subkey', 'valname')
value = winqueryreg('rootkey', 'subkey')
```

Description

`valnames = winqueryreg('name', 'rootkey', 'subkey')` returns all value names in `rootkey\subkey` of Microsoft Windows operating system registry to a cell array of strings. The first argument is the literal quoted string, `'name'`.

`value = winqueryreg('rootkey', 'subkey', 'valname')` returns the value for value name `valname` in `rootkey\subkey`.

If the value retrieved from the registry is a string, `winqueryreg` returns a string. If the value is a 32-bit integer, `winqueryreg` returns the value as an integer of the MATLAB software type `int32`.

`value = winqueryreg('rootkey', 'subkey')` returns a value in `rootkey\subkey` that has no value name property.

Note The literal **`name`** argument and the `rootkey` argument are case-sensitive. The `subkey` and `valname` arguments are not.

Tips

This function works only for the following registry value types:

- strings (REG_SZ)
- expanded strings (REG_EXPAND_SZ)
- 32-bit integer (REG_DWORD)

Examples

Example 1

Get the value of `CLSID` for the MATLAB sample Microsoft COM control `mwsampctrl.2`:

```
winqueryreg 'HKEY_CLASSES_ROOT' 'mwsamp.mwsampctrl.2\clsid'
```

```
ans =
   {5771A80A-2294-4CAC-A75B-157DCDDD3653}
```

Example 2

Get a list in variable mousechar for registry subkey Mouse, which is under subkey Control Panel, which is under root key HKEY_CURRENT_USER.

```
mousechar = winqueryreg('name', 'HKEY_CURRENT_USER', ...
   'control panel\mouse');
```

For each name in the mousechar list, get its value from the registry and then display the name and its value:

```
for k=1:length(mousechar)
   setting = winqueryreg('HKEY_CURRENT_USER', ...
      'control panel\mouse', mousechar{k});
   str = sprintf('%s = %s', mousechar{k}, num2str(setting));
   disp(str)
   end

ActiveWindowTracking = 0
DoubleClickHeight = 4
DoubleClickSpeed = 830
DoubleClickWidth = 4
MouseSpeed = 1
MouseThreshold1 = 6
MouseThreshold2 = 10
SnapToDefaultButton = 0
SwapMouseButtons = 0
```

Purpose

Determine whether file contains 1-2-3 WK1 worksheet

Note `wk1finfo` has been removed.

Syntax

```
[extens, typ] = wk1finfo(filename)
```

Description

`[extens, typ] = wk1finfo(filename)` returns the string 'WK1' in `extens`, and ' `1-2-3 Spreadsheet`' in `typ` if the file `filename` contains a readable worksheet. The `filename` input is a string enclosed in single quotes.

Examples

This example returns information on spreadsheet file `matA.wk1`:

```
[extens, typ] = wk1finfo('matA.wk1')

extens =
   WK1
typ =
    123 Spreadsheet
```

See Also

`xlsread` | `xlswrite` | `dlmread` | `dlmwrite`

wk1read

Purpose

Read Lotus 1-2-3 WK1 spreadsheet file into matrix

Note `wk1read` has been removed.

Syntax

```
M = wk1read(filename)
M = wk1read(filename,r,c)
M = wk1read(filename,r,c,range)
```

Description

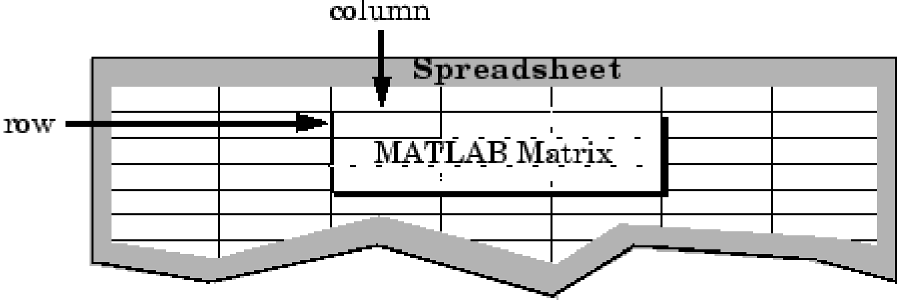
`M = wk1read(filename)` reads a Lotus1-2-3 WK1 spreadsheet file into the matrix `M`. The `filename` input is a string enclosed in single quotes.

`M = wk1read(filename,r,c)` starts reading at the row-column cell offset specified by `(r,c)`. `r` and `c` are zero based so that `r=0`, `c=0` specifies the first value in the file.

`M = wk1read(filename,r,c,range)` reads the range of values specified by the parameter `range`, where `range` can be

- A four-element vector specifying the cell range in the format

```
[upper_left_row upper_left_col lower_right_row lower_right_col]
```



- A cell range specified as a string, for example, `'A1...C5'`
- A named range specified as a string, for example, `'Sales'`

Examples

Create a 8-by-8 matrix A and export it to Lotus spreadsheet matA.wk1:

```
A = [1:8; 11:18; 21:28; 31:38; 41:48; 51:58; 61:68; 71:78]
A =
     1     2     3     4     5     6     7     8
    11    12    13    14    15    16    17    18
    21    22    23    24    25    26    27    28
    31    32    33    34    35    36    37    38
    41    42    43    44    45    46    47    48
    51    52    53    54    55    56    57    58
    61    62    63    64    65    66    67    68
    71    72    73    74    75    76    77    78

wk1write('matA.wk1', A);
```

To read in a limited block of the spreadsheet data, specify the upper left row and column of the block using zero-based indexing:

```
M = wk1read('matA.wk1', 3, 2)
M =
    33    34    35    36    37    38
    43    44    45    46    47    48
    53    54    55    56    57    58
    63    64    65    66    67    68
    73    74    75    76    77    78
```

To select a more restricted block of data, you can specify both the upper left and lower right corners of the block you want imported. Read in a range of values from row 4, column 3 (defining the upper left corner) to row 6, column 6 (defining the lower right corner). Note that, unlike the second and third arguments, the range argument [4 3 6 6] is one-based:

```
M = wk1read('matA.wk1', 3, 2, [4 3 6 6])
M =
    33    34    35    36
    43    44    45    46
    53    54    55    56
```

See Also `xlsread`

Purpose

Write matrix to Lotus 1-2-3 WK1 spreadsheet file

Note `wk1write` has been removed.

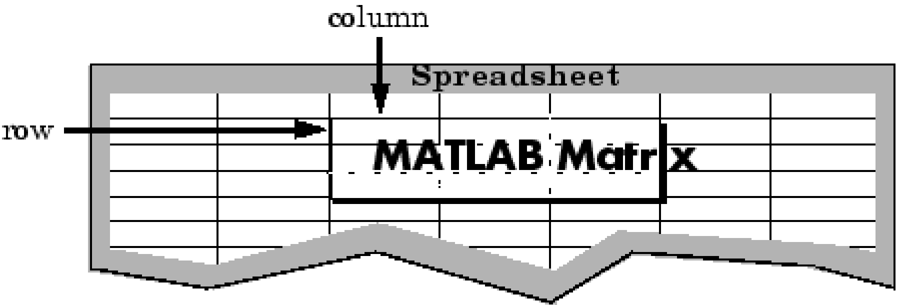
Syntax

```
wk1write(filename,M)
wk1write(filename,M,r,c)
```

Description

`wk1write(filename,M)` writes the matrix `M` into a Lotus1-2-3 WK1 spreadsheet file named `filename`. The `filename` input is a string enclosed in single quotes.

`wk1write(filename,M,r,c)` writes the matrix starting at the spreadsheet location `(r,c)`. `r` and `c` are zero based so that `r=0`, `c=0` specifies the first cell in the spreadsheet.



Examples

Write a 4-by-5 matrix A to spreadsheet file matA.wk1. Place the matrix with its upper left corner at row 2, column 3 using zero-based indexing:

```
A = [1:5; 11:15; 21:25; 31:35]
A =
      1      2      3      4      5
     11     12     13     14     15
     21     22     23     24     25
     31     32     33     34     35
```

```
wk1write('matA.wk1', A, 2, 3)

M = wk1read('matA.wk1')
M =
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     1     2     3     4     5
     0     0     0    11    12    13    14    15
     0     0     0    21    22    23    24    25
     0     0     0    31    32    33    34    35
```

See Also

dlmwrite | dlmread | xlswrite | xlsread

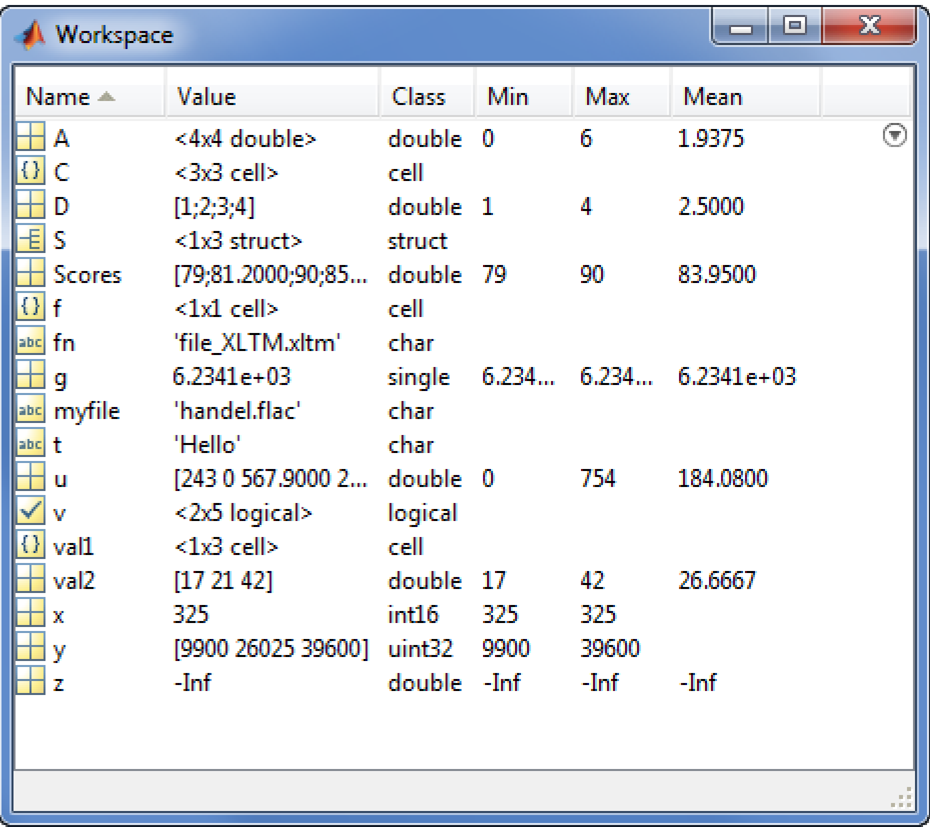
Purpose Open Workspace browser to manage workspace

Syntax

```
workspace
```

Description `workspace` displays the Workspace browser, a graphical user interface that allows you to view and manage the contents of the workspace in MATLAB. It provides a graphical representation of the `whos` display, and allows you to perform the equivalent of the `clear`, `load`, `open`, and `save` functions.

The Workspace browser also displays and automatically updates statistical calculations for each variable, which you can choose to show or hide.



You can edit a value directly in the Workspace browser for small numeric and character arrays. To see and edit a graphical representation of larger variables and for other classes, double-click the variable in the Workspace browser. The variable displays in the Variables editor, where you can view the full contents and make changes.

See Also

`openvar` | `who`

How To

- "What Is the MATLAB Workspace?"

Purpose

Write entire image

Syntax

```
tiffobj.write(imageData)
tiffobj.write(Y,Cb,Cr)
```

Description

`tiffobj.write(imageData)` writes imageData to TIFF file associated with the Tiff object, `tiffobj`. The `write` method breaks the data into strips or tiles, depending on the value of the `RowsPerStrip` tag, or the `TileLength` and `TileWidth` tags.

`tiffobj.write(Y,Cb,Cr)` writes the YCbCr component data to the TIFF file.

See Also

`Tiff.writeDirectory`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"

Tiff.writeDirectory

Purpose

Create new IFD and make it current IFD

Syntax

```
tiffobj.writeDirectory()
```

Description

`tiffobj.writeDirectory()` create a new image file directory (IFD) and makes it the current IFD. `Tiff` object methods operate on the current IFD. If you are creating a TIFF file that only contains one image, you do not need to use this method. With single-image TIFF files, just close the `Tiff` object to write data to the file.

Examples

Open a TIFF file for modification and create a new IFD in the file. `writeDirectory` makes the newly created IFD the current IFD. Replace the name `myfile.tif` with the name of a TIFF file on your MATLAB path.

```
t = Tiff('myfile.tif', 'r+');
dnum = t.currentDirectory();
t.writeDirectory();
dnum = t.currentDirectory();
```

References

This method corresponds to the `TIFFWriteDirectory` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff.write` | `Tiff.close`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

Purpose

Write data to specified strip

Syntax

```
tiffobj.writeEncodedStrip(stripNumber,imageData)
tiffobj.writeEncodedStrip(stripNumber,Y,Cb,Cr)
```

Description

`tiffobj.writeEncodedStrip(stripNumber,imageData)` writes the data in `imageData` to the strip specified by `stripNumber`. Strip identification numbers are one-based. If `imageData` has fewer bytes than fit into a strip, `writeEncodedStrip` silently pads the strip. If `imageData` has more bytes than fit into a strip, `writeEncodedStrip` issues a warning and truncates the data. To determine the size of a strip, view the value of the `RowsPerStrip` tag.

`tiffobj.writeEncodedStrip(stripNumber,Y,Cb,Cr)` writes the YCbCr component data to the specified tile. You must set the `YCbCrSubSampling` tag.

Examples

Open a `Tiff` object for modification. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path. The folder in which you run the example must be writable.

```
t = Tiff('myfile.tif', 'r+');

if ~t.isTiled()
    width = t.getTag('ImageWidth');
   height = t.getTag('RowsPerStrip');
   numSamples = t.getTag('SamplesPerPixel');
   imageData = zeros(height,width,numSamples,'uint8');
   t.writeEncodedStrip(1,imageData);
end
```

References

This method corresponds to the `TIFFWriteEncodedStrip` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff.writeEncodedTile`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

Purpose

Write data to specified tile

Syntax

```
tiffobj.writeEncodedTile(tileNumber,imageData)
tiffobj.writeEncodedTile(tileNumber,Y,Cb,Cr)
```

Description

`tiffobj.writeEncodedTile(tileNumber,imageData)` writes the data in `imageData` to the tile specified by `tileNumber`. Tile identification numbers are one-based. If `imageData` has fewer bytes than fit into a tile, `writeEncodedTile` silently pads the tile. If `imageData` has more bytes than fit into a tile, `writeEncodedTile` issues a warning and truncates the data. To determine the size of a tile, view the value of the `tileLength` and `tileWidth` tags.

`tiffobj.writeEncodedTile(tileNumber,Y,Cb,Cr)` writes the YCbCr component data to the specified tile. You must set the `YCbCrSubSampling` tags.

Examples

Open a TIFF file for modification. Replace `myfile.tif` with the name of a TIFF file on your MATLAB path.

```
t = Tiff('myfile.tif', 'r+');

if t.isTiled()
   width = t.getTag('tileWidth');
   height = t.getTag('tileLength');
   numSamples = t.getTag('SamplesPerPixel');
   imageData = zeros(height,width,numSamples,'uint8');
   t.writeEncodedTile(1,imageData);
end
```

References

This method corresponds to the `TIFFWriteEncodedTile` function in the LibTIFF C API. To use this method, you must be familiar with the TIFF specification and technical notes. View this documentation at `LibTIFF - TIFF Library and Utilities`.

See Also

`Tiff.writeEncodedStrip`

Tutorials

- "Exporting Image Data and Metadata to TIFF Files"
- "Reading Image Data and Metadata from TIFF Files"

Purpose

Write video data to file

Syntax

```
writeVideo(writerObj,img)
writeVideo(writerObj,images)
writeVideo(writerObj,frame)
writeVideo(writerObj,mov)
```

Description

`writeVideo(writerObj,img)` writes data from an image to a video file.

`writeVideo(writerObj,images)` writes a sequence of color images to a video file.

`writeVideo(writerObj,frame)` writes a frame to the video file associated with `writerObj`.

`writeVideo(writerObj,mov)` writes a MATLAB movie to a video file. `mov` is an array of frame structures.

You must call `open(writerObj)` before calling `writeVideo`.

Input Arguments

writerObj

`VideoWriter` object created by the `VideoWriter` function.

img

When creating AVI or MPEG-4 files, `img` is an array of `single`, `double`, or `uint8` values representing a grayscale or RGB color image, which `writeVideo` writes as an RGB video frame. Data of type `single` or `double` must be in the range `[0,1]`, except when writing Indexed AVI files.

When creating Motion JPEG 2000 files, `img` is an array of `uint8`, `int8`, `uint16`, or `int16` values representing a monochrome or RGB color image.

For grayscale, monochrome, or indexed data, `img` must be two-dimensional: height-by-width. For color data that is not indexed, `img` is three-dimensional: height-by-width-by-3. The height and width must be consistent for all frames within a file. For more information, see “Image Types”.

VideoWriter.writeVideo

images

Four-dimensional array of grayscale (height-by-width-by-1-by-frames) or RGB (height-by-width-by-3-by-frames) images.

frame

Structure typically returned by the `getframe` function that contains two fields: `cdata` and `colormap`. If the `colormap` is not empty, `writeVideo` expects a two-dimensional (height-by-width) array `cdata`. The height and width must be consistent for all frames within a file. `colormap` can contain a maximum of 256 entries.

The `profile` of `writerObj` and the size of `cdata` determine how the `writeVideo` method uses `frame`.

`profile` of `VideoWriter` object	Size of `cdata`	Behavior of `writeVideo`
`'Indexed AVI'` or `'Grayscale AVI'`	2-dimensional (height-by-width)	Use frame as provided. For `'Grayscale AVI'`, `colormap` should be empty.
	3-dimensional (height-by-width-by-3)	Error
All other profiles	2-dimensional (height-by-width)	Construct RGB image frames using the `colormap` field
	3-dimensional (height-by-width-by-3)	`Colormap` field ignored. Construct RGB image frames using the `cdata` field

mov

1-by-F array of frame structures, where F is the number of images. Each frame structure contains fields cdata and colormap.

Examples

AVI File from Animation

Write a sequence of frames to a compressed AVI file, peaks.avi.

Prepare the new file.

```
writerObj = VideoWriter('peaks.avi');
open(writerObj);
```

Generate initial data and set axes and figure properties.

```
Z = peaks; surf(Z);
axis tight
set(gca,'nextplot','replacechildren');
set(gcf,'Renderer','zbuffer');
```

Setting the Renderer property to zbuffer or Painters works around limitations of getframe with the OpenGL renderer on some Windows systems.

Create a set of frames and write each frame to the file.

```
for k = 1:20
   surf(sin(2*pi*k/20)*Z,Z)
   frame = getframe;
   writeVideo(writerObj,frame);
end

close(writerObj);
```

MPG to AVI Conversion

Convert an example file, xylophone.mpg, to an uncompressed AVI file:

Create objects to read and write the video, and open the AVI file for writing.

```
readerObj = VideoReader('xylophone.mpg');
writerObj = VideoWriter('transcoded_xylophone.avi', ...
                        'Uncompressed AVI');

open(writerObj);
```

Read and write each frame.

```
for k = 1:readerObj.NumberOfFrames
   img = read(readerObj,k);
   writeVideo(writerObj,img);
end

close(writerObj);
```

See Also

close | getframe | VideoReader | VideoWriter | movie2avi | open

Purpose

Label *x*-axis

Syntax

```
xlabel(str)
xlabel(str,Name,Value)

xlabel(axes_handle,___)

h = xlabel(___)
```

Description

`xlabel(str)` labels the *x*-axis of the current axes with the string, `str`. Each axes graphics object has one predefined *x*-axis label. Reissuing the `xlabel` command causes the new label to replace the old label. Labels appear beneath the axis in a two-dimensional view and to the side or beneath the axis in a three-dimensional view.

`xlabel(str,Name,Value)` additionally specifies the text object properties using one or more `Name,Value` pair arguments.

`xlabel(axes_handle,___)` adds the label to the axes specified by `axes_handle`. This syntax allows you to specify the axes to which to add a label. `axes_handle` can precede any of the input argument combinations in the previous syntaxes.

`h = xlabel(___)` returns the handle to the text object used as the *x*-axis label. The handle is useful when making future modifications to the label.

Input Arguments

str - Text to display as x-axis label

string

Text to display as the *x*-axis label, specified as a string or the name of a function that returns a string.

Example: `'myLabel'`

axes_handle - Axes handle

Axes handle, which is the reference to an axes object. Use the gca function to get the handle to the current axes, for example, axes_handle = gca;.

Name-Value Pair Arguments

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

Example: 'Color','red','FontSize',12 adds an x-axis label with red, 12-point font.

In addition to the following, you can specify other text object properties using Name,Value pair arguments. See Text Properties.

'Color' - Text color

[0 0 0] (black) (default) | 3-element RGB vector | string

Text color, specified as the comma-separated pair consisting of 'Color' and a 3-element RGB vector or a string containing the short or long name of the color. The RGB vector is a 3-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0 1].

The following table lists the predefined colors and their RGB equivalents.

RGB Value	Short Name	Long Name
[1 1 0]	y	yellow
[1 0 1]	m	magenta
[0 1 1]	c	cyan
[1 0 0]	r	red
[0 1 0]	g	green
[0 0 1]	b	blue

RGB Value	Short Name	Long Name
[1 1 1]	w	white
[0 0 0]	k	black

Example: `'Color',[0 1 0]`

Example: `'Color','green'`

'FontAngle' - Character slant

`'normal'` (default) | `'italic'` | `'oblique'`

Character slant, specified as the comma-separated pair consisting of `'FontAngle'` and one of these values: `'normal'`, `'italic'`, or `'oblique'`. MATLAB uses the `FontAngle` property to select a font from those available on your particular system. Generally, setting this property to `italic` or `oblique` selects a slanted font.

Example: `'FontAngle','italic'`

'FontName' - Font name

`'Helvetica'` (default) | string | `'FixedWidth'`

Font name, specified as the comma-separated pair consisting of `'FontName'` and a string. The string specifies the name of the font to use for the text object. To display and print properly, this must be a font that your system supports.

To use a fixed-width font that looks good in any locale, use the case-sensitive string `'FixedWidth'`. This eliminates the need to hard-code the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding.

Example: `'FontName','Courier'`

'FontSize' - Font size

10 points (default) | scalar

Font size, specified as the comma-separated pair consisting of 'FontSize' and a scalar in units determined by the FontUnits property. The default value for FontUnits is points.

Example: 'FontSize',12.5

'FontUnits' - Font size units
'points' (default) | 'normalized' | 'inches' | 'centimeters' | 'pixels'

Font size units, specified as the comma-separated pair consisting of 'FontUnits' and one of the following strings:

- 'points'
- 'normalized'
- 'inches'
- 'centimeters'
- 'pixels'

When the value of FontUnits is 'normalized', MATLAB interprets the value of FontSize as a fraction of the height of the parent axes. When you resize the axes, MATLAB modifies the screen FontSize accordingly. points, inches, centimeters, and pixels are absolute units. 1 point = $^1/_{72}$ inch

---

Note When setting both the FontSize and the FontUnits, you must set the FontUnits property first so that MATLAB can correctly interpret the specified FontSize. For example, to set the font size to 0.3 inches, call 'FontUnits','inches','FontSize',0.3 in the argument list.

'FontWeight' - Weight of text characters
'normal' (default) | 'bold' | 'light' | 'demi'

Weight of text characters, specified as the comma-separated pair consisting of 'FontWeight' and one of the following strings:

- `'normal'`
- `'bold'`
- `'light'`
- `'demi'`

MATLAB uses the `FontWeight` property to select a font from those available on your particular system. Generally, setting this property to `'bold'` or `'demi'` causes MATLAB to use a bold font.

Example: `'FontWeight','bold'`

'Interpreter' - Character interpretation

`'tex'` (default) | `'latex'` | `'none'`

Character interpretation, specified as the comma-separated pair consisting of `'Interpreter'` and one of the following strings.

Interpreter value	Result
`'tex'`	Supports a subset of plain TeX markup language. See the `String` property for a list of supported TeX instructions.
`'latex'`	Supports a basic subset of the LaTeX markup language.
`'none'`	Interprets all characters as literal characters.

Example: `'Interpreter','latex'`

Output Arguments

h - Handle to text object used as x-axis label

Handle to the text object used as the *x*-axis label. This is a unique identifier, which you can use to query and modify the properties of the label.

Examples

Label x-axis with String

```
figure
plot((1:10).^2)
xlabel('Population')
```

MATLAB displays `Population` beneath the *x*-axis.

Label x-axis with Numeric Input

```
figure
plot((1:10).^2)
xlabel(123)
```

MATLAB displays `123` beneath the *x*-axis.

Create Multiline Label

Create a multiline label using a multiline cell array.

```
figure
plot((1:10).^2)
xlabel({date;'Population';'in Years'})
```

Create Label and Set Font Properties

Use `Name,Value` pair arguments to set the font size, font weight, and text color properties of the *x*-axis label.

```
figure
plot((1:10).^2)
xlabel('Population','FontSize',12,'FontWeight','bold','Color','r')
```

`'FontSize',12` displays the label text in 12-point font. `'FontWeight','bold'` makes the text bold. `'Color','r'` sets the text color to red.

Label x-Axis of Specific Axes

Create two subplots and return the handles to the axes objects, `s(1)` and `s(2)`.

```
figure
s(1) = subplot(2,1,1);
plot((1:10).^2)
s(2) = subplot(2,1,2);
plot((1:10).^3)
```

Label the *x*-axis of the top plot by referring to its axes handle, `s(1)`.

```
xlabel(s(1),'Population')
```

Label x-axis and Return Object Handle

Label the *x*-axis and return the handle to the text object used as the label.

```
figure
plot((1:10).^2)
str = 'Population';
h = xlabel(str);
```

MATLAB returns the object handle in the output variable, `h`.

Set the color of the label to red, using the object handle.

```
set(h,'Color','red')
```

See Also

`strings` | `ylabel` | `zlabel` | `text` | `title`

Concepts

- "Adding Axis Labels to Graphs"

Text Properties

ylabel

Purpose

Label *y*-axis

Syntax

```
ylabel(str)
ylabel(str,Name,Value)

ylabel(axes_handle,___)

h = ylabel(___)
```

Description

`ylabel(str)` labels the *y*-axis of the current axes with the string, `str`. Every axes has one predefined *y*-axis label. Reissuing the `ylabel` command causes the new label to replace the old label. Labels appear beside the axis in a two-dimensional view and to the side or in front of the axis in a three-dimensional view.

`ylabel(str,Name,Value)` additionally specifies the text object properties using one or more `Name,Value` pair arguments.

`ylabel(axes_handle,___)` adds the label to the axes specified by `axes_handle`. This syntax allows you to specify the axes to which to add a label. `axes_handle` can precede any of the input argument combinations in the previous syntaxes.

`h = ylabel(___)` returns the handle to the text object used as the *y*-axis label. The handle is useful when making future modifications to the label.

Input Arguments

str - Text to display as y-axis label

string

Text to display as the *y*-axis label, specified as a string or the name of a function that returns a string.

Example: `'myLabel'`

axes_handle - Axes handle

Axes handle, which is the reference to an axes object. Use the gca function to get the handle to the current axes, for example, axes_handle = gca;.

Name-Value Pair Arguments

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

Example: 'Color','red','FontSize',12 adds a y-axis label with red, 12-point font.

In addition to the following, you can specify other text object properties using Name,Value pair arguments. See Text Properties.

'Color' - Text color

[0 0 0] (black) (default) | 3-element RGB vector | string

Text color, specified as the comma-separated pair consisting of 'Color' and a 3-element RGB vector or a string containing the short or long name of the color. The RGB vector is a 3-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0 1].

The following table lists the predefined colors and their RGB equivalents.

RGB Value	Short Name	Long Name
[1 1 0]	y	yellow
[1 0 1]	m	magenta
[0 1 1]	c	cyan
[1 0 0]	r	red
[0 1 0]	g	green
[0 0 1]	b	blue

RGB Value	Short Name	Long Name
[1 1 1]	w	white
[0 0 0]	k	black

Example: 'Color',[0 1 0]

Example: 'Color','green'

'FontAngle' - Character slant

'normal' (default) | 'italic' | 'oblique'

Character slant, specified as the comma-separated pair consisting of 'FontAngle' and one of these values: 'normal', 'italic', or 'oblique'. MATLAB uses the FontAngle property to select a font from those available on your particular system. Generally, setting this property to italic or oblique selects a slanted font.

Example: 'FontAngle','italic'

'FontName' - Font name

'Helvetica' (default) | string | 'FixedWidth'

Font name, specified as the comma-separated pair consisting of 'FontName' and a string. The string specifies the name of the font to use for the text object. To display and print properly, this must be a font that your system supports.

To use a fixed-width font that looks good in any locale, use the case-sensitive string 'FixedWidth'. This eliminates the need to hard-code the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding.

Example: 'FontName','Courier'

'FontSize' - Font size

10 points (default) | scalar

Font size, specified as the comma-separated pair consisting of 'FontSize' and a scalar in units determined by the FontUnits property. The default value for FontUnits is points.

Example: 'FontSize',12.5

'FontUnits' - Font size units

'points' (default) | 'normalized' | 'inches' | 'centimeters' | 'pixels'

Font size units, specified as the comma-separated pair consisting of 'FontUnits' and one of the following strings:

- 'points'
- 'normalized'
- 'inches'
- 'centimeters'
- 'pixels'

When the value of FontUnits is 'normalized', MATLAB interprets the value of FontSize as a fraction of the height of the parent axes. When you resize the axes, MATLAB modifies the screen FontSize accordingly. points, inches, centimeters, and pixels are absolute units. 1 point = $^1/_{72}$ inch

---

Note When setting both the FontSize and the FontUnits, you must set the FontUnits property first so that MATLAB can correctly interpret the specified FontSize. For example, to set the font size to 0.3 inches, call 'FontUnits','inches','FontSize',0.3 in the argument list.

'FontWeight' - Weight of text characters

'normal' (default) | 'bold' | 'light' | 'demi'

Weight of text characters, specified as the comma-separated pair consisting of 'FontWeight' and one of the following strings:

- 'normal'
- 'bold'
- 'light'
- 'demi'

MATLAB uses the FontWeight property to select a font from those available on your particular system. Generally, setting this property to 'bold' or 'demi' causes MATLAB to use a bold font.

Example: 'FontWeight','bold'

'Interpreter' - Character interpretation

'tex' (default) | 'latex' | 'none'

Character interpretation, specified as the comma-separated pair consisting of 'Interpreter' and one of the following strings.

Interpreter value	Result
'tex'	Supports a subset of plain TeX markup language. See the String property for a list of supported TeX instructions.
'latex'	Supports a basic subset of the LaTeX markup language.
'none'	Interprets all characters as literal characters.

Example: 'Interpreter','latex'

Output Arguments

h - Handle to text object used as y-axis label

Handle to the text object used as the *y*-axis label. This is a unique identifier, which you can use to query and modify the properties of the label.

Examples

Label y-axis with String

```
figure
plot((1:10).^2)
ylabel('Population')
```

MATLAB displays `Population` beside the *y*-axis.

Label y-axis with Numeric Input

```
figure
plot((1:10).^2)
ylabel(123)
```

MATLAB displays `123` beside the *y*-axis.

Create Multiline Label

Create a multiline label using a multiline cell array.

```
figure
plot((1:10).^2)
ylabel({2010;'Population';'in Years'})
```

Create Label and Set Font Properties

Use `Name,Value` pairs to set the font size, font weight, and text color properties of the *y*-axis label.

```
figure
plot((1:10).^2)
ylabel('Population','FontSize',12,'FontWeight','bold','Color','r')
```

`'FontSize',12` displays the label text in 12-point font. `'FontWeight','bold'` makes the text bold. `'Color','r'` sets the text color to red.

Label y-Axis of Specific Axes

Create two subplots and return the handles to the axes objects, `s(1)` and `s(2)`.

```
figure
s(1) = subplot(2,1,1);
plot((1:10).^2)
s(2) = subplot(2,1,2);
plot((1:10).^3)
```

Label the *y*-axis of the top plot by referring to its axes handle, `s(1)`.

```
ylabel(s(1),'Population')
```

Label y-axis and Return Object Handle

Label the *y*-axis and return the handle to the text object used as the label.

```
figure
plot((1:10).^2)
str = 'Population';
h = ylabel(str);
```

MATLAB returns the object handle in the output variable, `h`.

Set the color of the label to red, using the object handle.

```
set(h,'Color','red')
```

See Also

`strings` | `xlabel` | `zlabel` | `text` | `title`

Concepts

- "Adding Axis Labels to Graphs"

Text Properties

Purpose

Label *z*-axis

Syntax

```
zlabel(str)
zlabel(str,Name,Value)

zlabel(axes_handle,___)

h = zlabel(___)
```

Description

`zlabel(str)` labels the *z*-axis of the current axes with the string, `str`. Every axes has one predefined *z*-axis label. Reissuing the `zlabel` command causes the new label to replace the old label.

`zlabel(str,Name,Value)` additionally specifies the text object properties using one or more `Name,Value` pair arguments.

`zlabel(axes_handle,___)` adds the label to the axes specified by `axes_handle`. This syntax allows you to specify the axes to which to add a label. `axes_handle` can precede any of the input argument combinations in the previous syntaxes.

`h = zlabel(___)` returns the handle to the text object used as the *z*-axis label. The handle is useful when making future modifications to the label.

Input Arguments

str - Text to display as z-axis label

string

Text to display as the *z*-axis label, specified as a string or the name of a function that returns a string.

Example: `'myLabel'`

axes_handle - Axes handle

Axes handle, which is the reference to an axes object. Use the gca function to get the handle to the current axes, for example, axes_handle = gca;.

Name-Value Pair Arguments

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

Example: 'Color','red','FontSize',12 adds an z-axis label with red, 12-point font.

In addition to the following, you can specify other text object properties using Name,Value pair arguments. See Text Properties.

'Color' - Text color

[0 0 0] (black) (default) | 3-element RGB vector | string

Text color, specified as the comma-separated pair consisting of 'Color' and a 3-element RGB vector or a string containing the short or long name of the color. The RGB vector is a 3-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0 1].

The following table lists the predefined colors and their RGB equivalents.

RGB Value	Short Name	Long Name
[1 1 0]	y	yellow
[1 0 1]	m	magenta
[0 1 1]	c	cyan
[1 0 0]	r	red
[0 1 0]	g	green
[0 0 1]	b	blue

RGB Value	Short Name	Long Name
[1 1 1]	w	white
[0 0 0]	k	black

Example: `'Color',[0 1 0]`

Example: `'Color','green'`

'FontAngle' - Character slant

`'normal'` (default) | `'italic'` | `'oblique'`

Character slant, specified as the comma-separated pair consisting of `'FontAngle'` and one of these values: `'normal'`, `'italic'`, or `'oblique'`. MATLAB uses the `FontAngle` property to select a font from those available on your particular system. Generally, setting this property to `italic` or `oblique` selects a slanted font.

Example: `'FontAngle','italic'`

'FontName' - Font name

`'Helvetica'` (default) | string | `'FixedWidth'`

Font name, specified as the comma-separated pair consisting of `'FontName'` and a string. The string specifies the name of the font to use for the text object. To display and print properly, this must be a font that your system supports.

To use a fixed-width font that looks good in any locale, use the case-sensitive string `'FixedWidth'`. This eliminates the need to hard-code the name of a fixed-width font, which might not display text properly on systems that do not use ASCII character encoding.

Example: `'FontName','Courier'`

'FontSize' - Font size

10 points (default) | scalar

Font size, specified as the comma-separated pair consisting of `'FontSize'` and a scalar in units determined by the `FontUnits` property. The default value for `FontUnits` is `points`.

Example: `'FontSize',12.5`

'FontUnits' - Font size units
`'points'` (default) | `'normalized'` | `'inches'` | `'centimeters'` | `'pixels'`

Font size units, specified as the comma-separated pair consisting of `'FontUnits'` and one of the following strings:

- `'points'`
- `'normalized'`
- `'inches'`
- `'centimeters'`
- `'pixels'`

When the value of `FontUnits` is `'normalized'`, MATLAB interprets the value of `FontSize` as a fraction of the height of the parent axes. When you resize the axes, MATLAB modifies the screen `FontSize` accordingly. `points`, `inches`, `centimeters`, and `pixels` are absolute units. 1 point = $^1/_{72}$ inch

---

Note When setting both the `FontSize` and the `FontUnits`, you must set the `FontUnits` property first so that MATLAB can correctly interpret the specified `FontSize`. For example, to set the font size to 0.3 inches, call `'FontUnits','inches','FontSize',0.3` in the argument list.

'FontWeight' - Weight of text characters
`'normal'` (default) | `'bold'` | `'light'` | `'demi'`

Weight of text characters, specified as the comma-separated pair consisting of `'FontWeight'` and one of the following strings:

- 'normal'
- 'bold'
- 'light'
- 'demi'

MATLAB uses the FontWeight property to select a font from those available on your particular system. Generally, setting this property to 'bold' or 'demi' causes MATLAB to use a bold font.

Example: 'FontWeight','bold'

'Interpreter' - Character interpretation

'tex' (default) | 'latex' | 'none'

Character interpretation, specified as the comma-separated pair consisting of 'Interpreter' and one of the following strings.

Interpreter value	Result
'tex'	Supports a subset of plain TeX markup language. See the String property for a list of supported TeX instructions.
'latex'	Supports a basic subset of the LaTeX markup language.
'none'	Interprets all characters as literal characters.

Example: 'Interpreter','latex'

Output Arguments

h - Handle to text object used as z-axis label

Handle to the text object used as the *z*-axis label. This is a unique identifier, which you can use to query and modify the properties of the label.

Examples

Label z-axis with String

```
figure
surf(peaks)
zlabel('Height')
```

MATLAB displays `Height` beside the z-axis.

Label z-axis with Numeric Input

```
figure
surf(peaks)
zlabel(123)
```

MATLAB displays `123` beside the z-axis.

Create Multiline Label

Create a multiline label using a multiline cell array.

```
figure
surf(peaks)
zlabel({'First Line';'Second Line'})
```

Create Label and Set Font Properties

Use `Name,Value` pairs to set the font size, font weight, and text color properties of the z-axis label.

```
figure
surf(peaks)
zlabel('Elevation','FontSize',12,'FontWeight','bold','Color','r');
```

`'FontSize',12` displays the label text in 12-point font. `'FontWeight','bold'` makes the text bold. `'Color','r'` sets the text color to red.

Label z-Axis of Specific Axes

Create two subplots and return the handles to the axes objects, `s(1)` and `s(2)`.

```
figure
s(1) = subplot(2,1,1);
surf(peaks(30))
s(2) = subplot(2,1,2);
surf(peaks(45))
```

Label the *z*-axis of each plot by referring to the axes handles, `s(1)` and `s(2)`.

```
zlabel(s(1),'Height1')
zlabel(s(2),'Height2')
```

Label z-axis and Return Object Handle

Label the *z*-axis and return the handle to the text object used as the label.

```
figure
surf(peaks)
str = 'Population Change';
h = zlabel(str);
```

MATLAB returns the object handle in the output variable, `h`.

Set the color of the label to red, using the object handle.

```
set(h,'Color','red')
```

See Also

`strings` | `ylabel` | `xlabel` | `text` | `title`

Concepts

- "Adding Axis Labels to Graphs"

Text Properties

xlim

Purpose

Set or query x-axis limits

Syntax

```
xlim
xlim([xmin xmax])
xlim('mode')
xlim('auto')
xlim('manual')
xlim(axes_handle,...)
```

Description

`xlim` with no arguments returns the limits of the current axes.

`xlim([xmin xmax])` sets the axis limits in the current axes to the specified values.

`xlim('mode')` returns the current value of the axis limits mode, which can be either `auto` (the default) or `manual`.

`xlim('auto')` sets the axis limit mode to `auto`.

`xlim('manual')` sets the axis limit mode to `manual`.

`xlim(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, it operates on the current axes.

Tips

`xlim` sets or queries values of the axes object `XLim`, and `XLimMode` property.

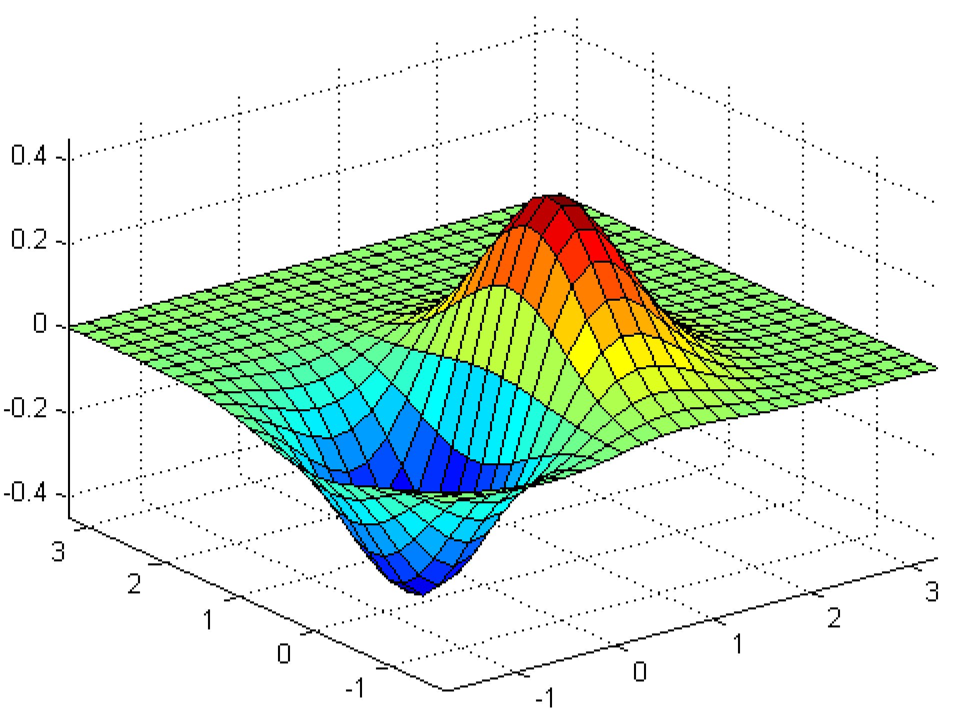
When the axis limit mode is set to `auto` (the default), MATLAB uses limits, which are round numbers, to span the range of the data being displayed. Setting a value for any of the limits also sets the corresponding mode to `manual`. If you set the limits on an existing graph and want to maintain these limits while adding more graphs, use the `hold` command.

Note High-level plotting functions like `plot` and `surf` reset both the modes and the limits.

Examples

This example illustrates how to set the *x*- and *y*-axis limits to match the actual range of the data, rather than the rounded values of `[-2 3]` for the *x*-axis, `[-2 4]` for the *y*-axis and `[-0.5 0.5]` for *z*-axis originally selected by MATLAB.

```
[x,y] = meshgrid([-1.75:.2:3.25]);
z = x.*exp(-x.^2-y.^2);
surf(x,y,z)
xlim([-1.75 3.25])
ylim([-1.75 3.25])
zlim([-0.45 0.45])
```



Alternatives

To control the upper and lower axis limits on a graph, use the Property Editor, one of the plotting tools . For details, see The Property Editor.

See Also

`ylim` | `zlim` | `axis`

How To

- `XLim`
- `YLim`

- `ZLim`
- Understanding Axes Aspect Ratio

ylim

Purpose

Set or query *y*-axis limits

Syntax

```
ylim
ylim([ymin ymax])
ylim('mode')
ylim('auto')
ylim('manual')
ylim(axes_handle,...)
```

Description

`ylim` with no arguments returns the limits of the current axes.

`ylim([ymin ymax])` sets the axis limits in the current axes to the specified values.

`ylim('mode')` returns the current value of the axis limits mode, which can be either `auto` (the default) or `manual`.

`ylim('auto')` sets the axis limit mode to `auto`.

`ylim('manual')` sets the axis limit mode to `manual`.

`ylim(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, it operates on the current axes.

Tips

`ylim` sets or queries values of the axes object `YLim`and `YLimMode` property.

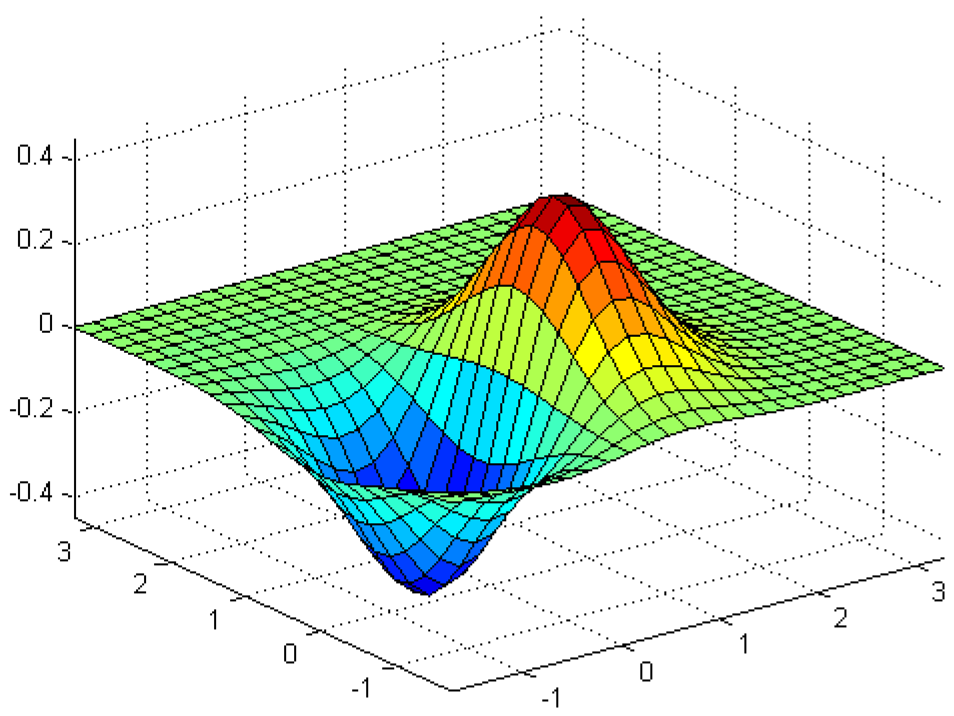
When the axis limit mode is set to `auto` (the default), MATLAB uses limits, which are round numbers, to span the range of the data being displayed. Setting a value for any of the limits also sets the corresponding mode to `manual`. If you set the limits on an existing graph and want to maintain these limits while adding more graphs, use the `hold` command.

Note High-level plotting functions like `plot` and `surf` reset both the modes and the limits.

Examples

This example illustrates how to set the *x*- and *y*-axis limits to match the actual range of the data, rather than the rounded values of `[-2 3]` for the *x*-axis and `[-2 4]` for the *y*-axis and `[-0.5 0.5]` for *z*-axis originally selected by MATLAB.

```
[x,y] = meshgrid([-1.75:.2:3.25]);
z = x.*exp(-x.^2-y.^2);
surf(x,y,z)
xlim([-1.75 3.25])
ylim([-1.75 3.25])
zlim([-0.45 0.45])
```



Alternatives

To control the upper and lower axis limits on a graph, use the Property Editor, one of the plotting tools . For details, see The Property Editor.

See Also

`xlim` | `zlim` | `axis`

How To

- `XLim`
- `YLim`

- `ZLim`
- Understanding Axes Aspect Ratio

zlim

Purpose

Set or query z-axis limits

Syntax

```
zlim
zlim([zmin zmax])
zlim('mode')
zlim('auto')
zlim('manual')
zlim(axes_handle,...)
```

Description

`zlim` with no arguments returns the limits of the current axes.

`zlim([zmin zmax])` sets the axis limits in the current axes to the specified values.

`zlim('mode')` returns the current value of the axis limits mode, which can be either `auto` (the default) or `manual`.

`zlim('auto')` sets the axis limit mode to `auto`.

`zlim('manual')` sets the axis limit mode to `manual`.

`zlim(axes_handle,...)` performs the set or query on the axes identified by the first argument, `axes_handle`. When you do not specify an axes handle, it operates on the current axes.

Tips

`zlim` sets or queries values of the axes object `ZLim` and `ZLimMode` property.

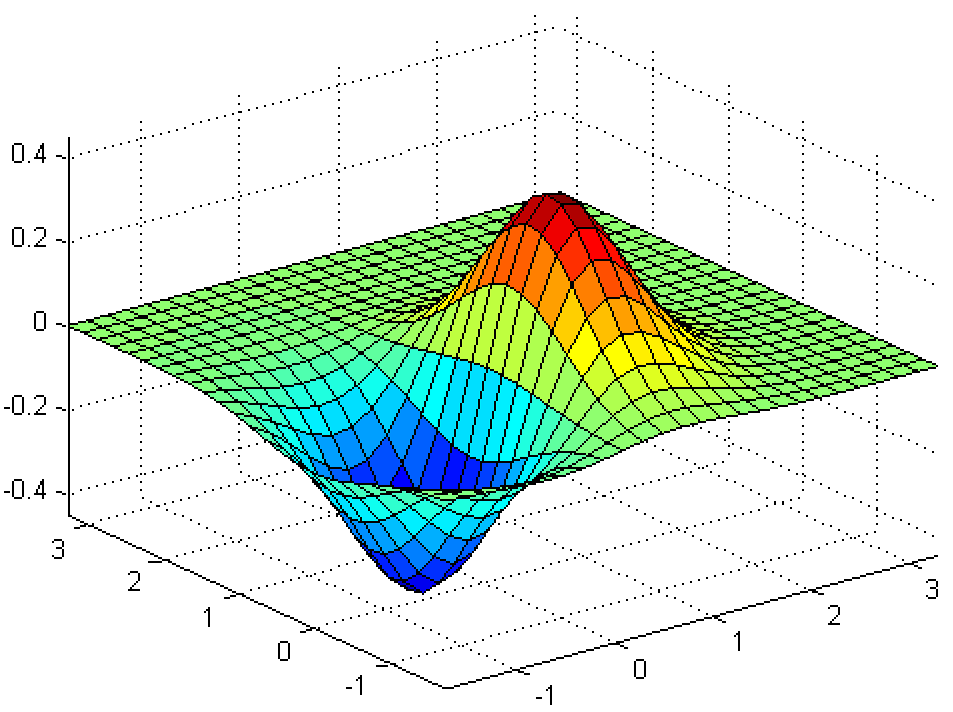
When the axis limit mode is set to `auto` (the default), MATLAB uses limits, which are round numbers, to span the range of the data being displayed. Setting a value for any of the limits also sets the corresponding mode to `manual`. If you set the limits on an existing graph and want to maintain these limits while adding more graphs, use the `hold` command.

Note High-level plotting functions like `plot` and `surf` reset both the modes and the limits.

Examples

This example illustrates how to set the *x*-, *y*-, and *z*-axis limits to match the actual range of the data, rather than the rounded values of `[-2 3]` for the *x*-axis ,`[-2 4]` for the *y*-axis and `[-0.5 0.5]` for *z*-axis originally selected by MATLAB.

```
[x,y] = meshgrid([-1.75:.2:3.25]);
z = x.*exp(-x.^2-y.^2);
surf(x,y,z)
xlim([-1.75 3.25])
ylim([-1.75 3.25])
zlim([-0.45 0.45])
```



Alternatives

To control the upper and lower axis limits on a graph, use the Property Editor, one of the plotting tools . For details, see The Property Editor.

See Also

`xlim` | `ylim` | `axis`

How To

- `XLim`
- `YLim`

- `ZLim`
- Understanding Axes Aspect Ratio

xlsfinfo

Purpose

Determine whether file contains Microsoft Excel spreadsheet

Syntax

```
status = xlsfinfo(filename)
[status,sheets] = xlsfinfo(filename)
[status,sheets,format] = xlsfinfo(filename)
```

Description

`status = xlsfinfo(filename)` returns a nonempty string when the specified file is in an Excel format that `xlsread` can read (for example, `'Microsoft Excel Spreadsheet'`). Otherwise, `status` is an empty string, `''`.

`[status,sheets] = xlsfinfo(filename)` returns a cell array of strings containing the names of each spreadsheet in the file.

`[status,sheets,format] = xlsfinfo(filename)` returns a string containing the format description that Excel returns for the file. On systems without Excel for Windows, `format` is an empty string, `''`.

Input Arguments

filename

String enclosed in single quotation marks that specifies the name of the file.

Output Arguments

status

If the file is in an Excel format that `xlsread` can read, `status` is a nonempty string. Otherwise, `status` is an empty string, `''`.

sheets

If the file is in an Excel format that `xlsread` can read, `sheets` is a `1`-by-`n` cell array of strings, where `n` is the number of worksheets. Each cell contains the name of a worksheet. If MATLAB cannot read a particular worksheet, the corresponding cell contains an error message.

If the file is not in a readable Excel format, `sheets` is a string containing an error message.

format

On Windows systems with Excel software, `format` is a string that contains the description that Excel returns for the file. For example:

`'xlOpenXMLWorkbook'`	Spreadsheet in XLSX format (Excel 2007 or later).
`'xlWorkbookNormal'` or `'xlExcel8'`	Spreadsheet in XLS format (compatible with Excel 97-2003).
`'xlCSV'`	File in comma-separated value (CSV) format.
`'xlHtml'` or `'xlWebArchive'`	Spreadsheet exported to HTML format.

On all other systems, `format` is an empty string, `''`.

Examples

Consider a hypothetical file with worksheets named `Sheet1`, `Income`, and `Expenses`. Get information about the file:

```
[typ, desc, fmt] = xlsfinfo('myaccount.xlsx')
```

This code returns:

```
typ =
    Microsoft Excel Spreadsheet

desc =
    'Sheet1'    'Income'    'Expenses'

fmt =
    xlOpenXMLWorkbook
```

Limitations

- `xlsfinfo` supports only 7-bit ASCII characters.

See Also

`xlsread` | `xlswrite`

xlsread

Purpose

Read Microsoft Excel spreadsheet file

Syntax

```
num = xlsread(filename)
num = xlsread(filename,sheet)
num = xlsread(filename,xlRange)
num = xlsread(filename,sheet,xlRange)
num = xlsread(filename,sheet,xlRange,'basic')
[num,txt,raw] = xlsread(___)

___ = xlsread(filename,-1)

[num,txt,raw,custom] = xlsread(filename,sheet,xlRange,'',
   functionHandle)
```

Description

`num = xlsread(filename)` reads data from the first worksheet in the Microsoft Excel spreadsheet file named `filename` and returns the numeric data in array `num`.

On Windows systems with Microsoft Excel software, `xlsread` reads any file format recognized by your version of Excel.

If your system does not have Excel for Windows, `xlsread` operates in `basic` import mode, and reads only XLS, XLSX, XLSM, XLTX, and XLTM files.

`num = xlsread(filename,sheet)` reads the specified worksheet.

`num = xlsread(filename,xlRange)` reads data from the specified range, `xlRange`, of the first worksheet in the file.

`num = xlsread(filename,sheet,xlRange)` reads from the specified `sheet` and range, `xlRange`.

`num = xlsread(filename,sheet,xlRange,'basic')` reads data from the spreadsheet in `basic` mode, the default on systems without Excel for

Windows. If you do not specify all the arguments, use empty strings as placeholders, for example, `num = xlsread(filename,'','','basic')`.

`[num,txt,raw] = xlsread(___)` additionally returns the text fields in cell array `txt`, and the unprocessed data (numbers and text) in cell array `raw` using any of the input arguments in the previous syntaxes. If `xlRange` is specified, leading blank rows and columns in the worksheet that precede rows and columns with data are returned in `raw`.

`___ = xlsread(filename,-1)` opens an Excel window to interactively select data. Select the worksheet, drag and drop the mouse over the range you want, and click **OK**. This syntax is supported only on Windows systems with Excel software.

`[num,txt,raw,custom] = xlsread(filename,sheet,xlRange,'',functionHandle)` reads from the spreadsheet, executes the function associated with `functionHandle` on the data, and returns the final results as numeric data in array `num`. The `xlsread` function optionally returns the text fields in cell array `txt`, the unprocessed data (numbers and text) in cell array `raw`, and the second output from the function associated with `functionHandle` in array `custom`. The `xlsread` function does not change the data stored in the spreadsheet. This syntax is supported only on Windows systems with Excel software.

Input Arguments

filename - Name of file to read

string

Name of the file to read, specified as a string. If you do not include an extension, `xlsread` searches for a file with the specified name and a supported Excel extension. `xlsread` can read data saved in files that are currently open in Excel for Windows.

Example: `'myFile.xlsx'`

Data Types

`char`

sheet - Worksheet to read

string | positive integer

Worksheet to read, specified as one of the following:

- String that contains the worksheet name. Cannot contain a colon (`:`). To determine the names of the sheets in a spreadsheet file, use `xlsfinfo`. For XLS files in `basic` mode, `sheet` is case sensitive.
- Positive integer that indicates the worksheet index. Not supported for XLS files in `basic` mode.

xlRange - Rectangular portion of the worksheet to read

string

Rectangular portion of the worksheet to read, specified as a string.

Specify `xlRange` using the syntax `'C1:C2'`, where `C1` and `C2` are two opposing corners that define the region to read. For example, `'D2:H4'` represents the 3-by-5 rectangular region between the two corners `D2` and `H4` on the worksheet. The `xlRange` input is not case sensitive, and uses Excel A1 reference style (see Excel help).

Range selection is not supported when reading XLS files in `basic` mode. In this case, use `''` in place of `xlRange`.

If you do not specify `sheet`, then `xlRange` must include both corners and a colon character, even for a single cell (such as `'D2:D2'`). Otherwise, `xlsread` interprets the input as a worksheet name (such as `'sales'` or `'D2'`).

If you specify `sheet`, then `xlRange`:

- Does not need to include a colon and opposite corner to describe a single cell.
- Can refer to a named range that you defined in the Excel file (see the Excel help).

Data Types

`char`

'basic' - Flag to request reading in `basic` mode

Flag to request reading in `basic` mode, which is the default for systems without Excel for Windows. In `basic` mode, `xlsread`:

- Reads XLS, XLSX, XLSM, XLTX, and XLTM files only.
- Does not support an `xlRange` input when reading XLS files. In this case, use `''` in place of `xlRange`.
- Does not support function handle inputs.
- Imports all dates as Excel serial date numbers. Excel serial date numbers use a different reference date than MATLAB date numbers.

functionHandle - Handle to a custom function

function handle

Handle to a custom function, starting with the symbol `@`. Supported only on Windows systems with Excel software. `xlsread` reads from the spreadsheet, executes your function on a copy of the data, and returns the final results. `xlsread` does not change the data stored in the spreadsheet.

When `xlsread` calls the custom function, it passes a range interface from the Excel application to provide access to the data. The custom function must include this interface both as an input and output argument. (See the Examples.)

Example: `@myFunctionHandle`

Output Arguments

num - Numeric data

matrix

Numeric data, returned as a matrix of `double` values. The array does not contain any information from header lines, or from outer rows or columns that contain nonnumeric data. Text data in inner spreadsheet rows and columns appear as `NaN` in the `num` output.

txt - Text data

cell array

Text data, returned as a cell array. Numeric values in inner spreadsheet rows and columns appear as empty strings, `''`, in `txt`.

For XLS files in `basic` import mode, the `txt` output contains empty strings, `''`, in place of leading columns of numeric data that precede text data in the spreadsheet. In all other cases, `txt` does not contain these additional columns.

Undefined values (such as `'#N/A'`) appear in the `txt` output as `'#N/A'`, except for XLS files in `basic` mode.

raw - Unprocessed data
cell array

Unprocessed data from the worksheet, returned as a cell array. Contains both numeric and text data.

On systems with Excel for Windows, undefined values (such as `'#N/A'`) appear in the `raw` output as `'ActiveX VT_ERROR:'`. For XLSX, XLSM, XLTX, and XLTM files on other systems, undefined values appear as `'#N/A'`.

custom - Second output of the function corresponding to `functionHandle`.

Second output of the function corresponding to `functionHandle`. The value and data type of `custom` are determined by the function.

Examples

Read Data from First Worksheet Into Numeric Array

Create an Excel file named `myExample.xlsx`.

```
values = {1, 2, 3 ; 4, 5, 'x' ; 7, 8, 9};
headers = {'First', 'Second', 'Third'};
xlswrite('myExample.xlsx', [headers; values]);
```

`Sheet1` of `myExample.xlsx` contains:

```
   First    Second    Third
       1         2        3
```

```
        4           5    x
        7           8         9
```

Read data from the first worksheet.

```
filename = 'myExample.xlsx';
A = xlsread(filename)
```

```
A =
     1     2     3
     4     5   NaN
     7     8     9
```

`xlsread` returns the numeric data in array `A`.

Read a Specific Range of Data

Read a specific range of data from the Excel file in the previous example.

```
filename = 'myExample.xlsx';
sheet = 1;
xlRange = 'B2:C3';

subsetA = xlsread(filename, sheet, xlRange)
```

```
subsetA =
     2     3
     5   NaN
```

Read a Column of Data

Read the second column of data from the Excel file in the first example.

```
filename = 'myExample.xlsx';

columnB = xlsread(filename,'B:B')
```

```
columnB =
     2
     5
```

```
     8
```

For better performance, specify the row numbers in the range, as shown in the previous example.

Request Numeric, Text, and Unprocessed Data

Request the numeric data, text, and a copy of the unprocessed (raw) data from the Excel file in the first example.

```
[ndata, text, alldata] = xlsread('myExample.xlsx')
```

```
ndata =
     1     2     3
     4     5   NaN
     7     8     9

text =
    'First'    'Second'    'Third'
    ''         ''          ''
    ''         ''          'x'

alldata =
    'First'    'Second'    'Third'
    [    1]    [     2]    [    3]
    [    4]    [     5]    'x'
    [    7]    [     8]    [    9]
```

`xlsread` returns numeric data in array `ndata`, text data in cell array `text`, and unprocessed data in cell array `alldata`.

Execute a Function on a Worksheet and Return Numeric Data

In the Editor, create a function to process data from a worksheet. In this case, set values outside the range `[-3, 3]` to `-3` or `3`.

```
function [Data] = setMinMax(Data)

  minval = -3; maxval = 3;
```

```
  for k = 1:Data.Count
    v = Data.Value{k};
    if v > maxval || v < minval
       if v > maxval
          Data.Value{k} = maxval;
       else
          Data.Value{k} = minval;
       end
    end
  end
```

In the Command Window, add data to `myExample.xlsx`.

```
misc = pi*gallery('normaldata',[10,3],1);
xlswrite('myExample.xlsx', misc, 'MyData');
```

Worksheet `MyData` contains the following values, which range from -6.6493 to 3.4845:

```
    2.7156   -6.1744    1.8064
    0.2959   -2.3383   -2.7210
   -2.6764   -1.7351   -6.6493
    2.7442   -2.5752   -3.0300
   -1.3761    3.4845    0.6683
   -1.3498   -1.9319    1.5014
   -3.4643   -0.8000    0.3162
    1.2448   -0.8477    0.9344
   -3.0314   -5.2527    1.7912
    0.5292   -5.8938   -5.1035
```

Read the data from the worksheet, and reset any values outside the range `[-3, 3]`. Specify the sheet name, but use `''` as placeholders for the `xlRange` and `'basic'` inputs.

```
trim = xlsread('myExample.xlsx','MyData','','',@setMinMax)

trim =
```

```
   2.7156   -3.0000    1.8064
   0.2959   -2.3383   -2.7210
  -2.6764   -1.7351   -3.0000
   2.7442   -2.5752   -3.0000
  -1.3761    3.0000    0.6683
  -1.3498   -1.9319    1.5014
  -3.0000   -0.8000    0.3162
   1.2448   -0.8477    0.9344
  -3.0000   -3.0000    1.7912
   0.5292   -3.0000   -3.0000
```

Request Custom Output

Execute a function on a worksheet and display the custom index output.

In the Editor, modify the function `setMinMax` from the previous example to return the indices of the changed elements (custom output).

```
function [Data, indices] = setMinMax(Data)

  minval = -3; maxval = 3;
  indices = [];

  for k = 1:Data.Count
    v = Data.Value{k};
    if v > maxval || v < minval
       if v > maxval
          Data.Value{k} = maxval;
       else
           Data.Value{k} = minval;
       end
       indices = [indices k];
    end
  end
```

Read the data from the worksheet `MyData`, and request the custom index output, `idx`.

```
[trim,txt,raw,idx] = xlsread('myExample.xlsx','MyData', '', '', @setMinMa
```

```
disp(idx)
```

```
7       9     11     15     19     20     23     24     30
```

Algorithms

- If you do not specify `xlRange`, then `xlsread` ignores leading blank rows and columns in the worksheet that precede your data.
- If you specify `xlRange`, then leading blank rows and columns in the worksheet that precede your data are returned in `raw`, but not in `num` or `txt`.
- When the specified `xlRange` overlaps merged cells:
  - On Windows systems with Excel, `xlsread` expands the range to include all merged cells.
  - On systems without Excel for Windows, `xlsread` returns data for the specified range only, with empty or `NaN` values for merged cells.
- `xlsread` imports formatted dates as strings (such as `'10/31/96'`), except in `basic` mode and on systems without Excel for Windows.

Limitations

- `xlsread` reads only 7-bit ASCII characters.
- `xlsread` does not support non-contiguous ranges.

See Also

`xlswrite` | `xlsfinfo` | `importdata` | `uiimport` | `function_handle`

Concepts

- “When to Convert Dates from Excel Files”

xlswrite

Purpose

Write Microsoft Excel spreadsheet file

Syntax

```
xlswrite(filename,A)
xlswrite(filename,A,sheet)
xlswrite(filename,A,xlRange)
xlswrite(filename,A,sheet,xlRange)

status = xlswrite( ___ )
[status,message] = xlswrite( ___ )
```

Description

`xlswrite(filename,A)` writes array `A` to the first worksheet in Excel file, `filename`, starting at cell `A1`.

`xlswrite(filename,A,sheet)` writes to the specified worksheet.

`xlswrite(filename,A,xlRange)` writes to the rectangular region specified by `xlRange` in the first worksheet of the file.

`xlswrite(filename,A,sheet,xlRange)` writes to the specified `sheet` and range, `xlRange`.

`status = xlswrite( ___ )` returns the status of the write operation, and can include any of the input arguments in previous syntaxes. When the operation is successful, status is `1`. Otherwise, status is `0`.

`[status,message] = xlswrite( ___ )` additionally returns any warning or error message generated by the write operation in structure `message`.

Input Arguments

filename - Name of file to write

string

Name of file to write, specified as a string.

If `filename` does not exist, `xlswrite` creates a file, determining the format based on the specified extension. To create a file compatible with Excel 97-2003 software, specify an extension of `.xls`. To create files in Excel 2007 formats, specify an extension of `.xlsx`, `.xlsb`, or `.xlsm`. If you do not specify an extension, `xlswrite` uses the default, `.xls`.

Example: `'myFile.xlsx'`

A - Data to write

matrix | cell array

Data to write, specified as a two-dimensional numeric or character array, or, if each cell contains a single element, a cell array.

If `A` is a cell array containing something other than a scalar numeric or a string, then `xlswrite` silently leaves the corresponding cell in the spreadsheet empty.

The maximum size of array `A` depends on the associated Excel version. For more information on Excel specifications and limits, see the Excel help.

Example: `[10,2,45;-32,478,50]`

Example: `{92.0,'Yes',45.9,'No'}`

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `cell`

sheet - Worksheet name

string | positive integer

Worksheet name, specified as one of the following:

- String that contains the worksheet name. Cannot contain a colon (`:`). To determine the names of the sheets in a spreadsheet file, use `xlsfinfo`.
- Positive integer that indicates the worksheet index.

If sheet does not exist, xlswrite adds a new sheet at the end of the worksheet collection. If sheet is an index larger than the number of worksheets, xlswrite appends empty sheets until the number of worksheets in the workbook equals sheet. In either case, xlswrite generates a warning indicating that it has added a new worksheet.

xlRange - Rectangular portion of the worksheet to write
string

Rectangular portion of the worksheet to write, specified as a string.

Specify xlRange using the syntax 'C1:C2', where C1 and C2 are two opposing corners that define the region to write. For example, 'D2:H4' represents the 3-by-5 rectangular region between the two corners D2 and H4 on the worksheet. The xlRange input is not case sensitive, and uses Excel A1 reference style (see Excel help). xlswrite does not recognize named ranges.

- If you do not specify sheet, then xlRange must include both corners and a colon character, even for a single cell (such as 'D2:D2'). Otherwise, xlswrite interprets the input as a worksheet name (such as 'D2').
- If you specify sheet, then xlRange can specify only the first cell (such as 'D2'). xlswrite writes input array A beginning at this cell.
- If xlRange is larger than the size of input array A, Excel software fills the remainder of the region with #N/A. If xlRange is smaller than the size of A, then xlswrite writes only the subset that fits into xlRange to the file.

Output Arguments

status - Status of the write operation
1 | 0

Status of the write operation, returned as either 1 (true) or 0 (false). When the write operation is successful, status is 1. Otherwise, status is 0.

message - Error or warning generated during the write operation

structure array

Error or warning generated during the write operation, returned as a structure array containing two fields:

`message`	Text of the warning or error message, returned as a string.
`identifier`	Message identifier, returned as a string.

Examples

Write Data to a Spreadsheet

Write a 7-element vector to an Excel file, `testdata.xlsx`.

```
filename = 'testdata.xlsx';
A = [12.7, 5.02, -98, 63.9, 0, -.2, 56];
xlswrite(filename,A)
```

Write Data to a Specific Sheet and Range in a Spreadsheet

Write mixed text and numeric data to an Excel file, `testdata.xlsx`, starting at cell `E1` of `Sheet2`.

```
filename = 'testdata.xlsx';
A = {'Time','Temperature'; 12,98; 13,99; 14,97};
sheet = 2;
xlRange = 'E1';
xlswrite(filename,A,sheet,xlRange)
```

Tips

- If your system has Microsoft Office 2003 software, but you want to create a file in an Excel 2007 format, install the Office 2007 Compatibility Pack.
- Excel and MATLAB can store dates as strings (such as `'10/31/96'`) or serial date numbers (such as `729329`). If your array `A` includes serial date numbers, convert the dates to strings using `datestr` before calling `xlswrite`. Alternatively, see Converting Dates.

- To write data to Excel files with custom formats (such as fonts or colors), access the Windows COM server directly using `actxserver` rather than `xlswrite`. For example, Technical Solution 1-QLD4K uses `actxserver` to establish a connection between MATLAB and Excel, writes data to a worksheet, and specifies the colors of the cells.

Algorithms

Excel converts `Inf` values to `65535`. MATLAB converts `NaN` values to empty cells.

If your system does not have Excel for Windows, or if the COM server (part of the typical installation of Excel) is unavailable, then the `xlswrite` function:

- Writes array `A` to a text file in comma-separated value (CSV) format.
- Ignores the `sheet` and `xlRange` arguments.
- Generates an error when input array `A` is a cell array.

See Also

`xlsread` | `xlsfinfo`

Purpose

Read XML document and return Document Object Model node

Syntax

```
DOMnode = xmlread(filename)
```

Description

`DOMnode = xmlread(filename)` reads the specified XML file and returns a Document Object Model node representing the document.

Tips

The display for a properly parsed document is `[#document:  null]`. For example,

```
xDoc = xmlread('info.xml')
```

returns

```
xDoc =
[#document: null]
```

Input Arguments

filename

String enclosed in single quotation marks that specifies the name of the local file or URL.

Output Arguments

DOMnode

Document Object Model node, as defined by the World Wide Web consortium. For more information, see "What Is an XML Document Object Model (DOM)?".

Examples

The root element in an XML file sometimes includes an `xsi:noNamespaceSchemaLocation` attribute. The value of this attribute is the name of the preferred schema file. Call the `getAttribute` method to get this value:

```
xDoc = xmlread(fullfile(matlabroot,'toolbox',...
               'matlab','general','info.xml'));

xRoot = xDoc.getDocumentElement;
schema = char(xRoot.getAttribute('xsi:noNamespaceSchemaLocation'))
```

This code returns:

```
schema =
http://www.mathworks.com/namespace/info/v1/info.xsd
```

Create functions that parse data from an XML file into a MATLAB structure array with fields `Name`, `Attributes`, `Data`, and `Children`:

```
function theStruct = parseXML(filename)
% PARSEXML Convert XML file to a MATLAB structure.
try
   tree = xmlread(filename);
catch
   error('Failed to read XML file %s.',filename);
end

% Recurse over child nodes. This could run into problems
% with very deeply nested trees.
try
   theStruct = parseChildNodes(tree);
catch
   error('Unable to parse XML file %s.',filename);
end


% ----- Local function PARSECHILDNODES -----
function children = parseChildNodes(theNode)
% Recurse over node children.
children = [];
if theNode.hasChildNodes
   childNodes = theNode.getChildNodes;
   numChildNodes = childNodes.getLength;
   allocCell = cell(1, numChildNodes);

   children = struct(             ...
      'Name', allocCell, 'Attributes', allocCell,    ...
```

```
        'Data', allocCell, 'Children', allocCell);

      for count = 1:numChildNodes
          theChild = childNodes.item(count-1);
          children(count) = makeStructFromNode(theChild);
      end
end

% ----- Local function MAKESTRUCTFROMNODE -----
function nodeStruct = makeStructFromNode(theNode)
% Create structure of node info.

nodeStruct = struct(                        ...
   'Name', char(theNode.getNodeName),       ...
   'Attributes', parseAttributes(theNode),  ...
   'Data', '',                              ...
   'Children', parseChildNodes(theNode));

if any(strcmp(methods(theNode), 'getData'))
   nodeStruct.Data = char(theNode.getData);
else
   nodeStruct.Data = '';
end

% ----- Local function PARSEATTRIBUTES -----
function attributes = parseAttributes(theNode)
% Create attributes structure.

attributes = [];
if theNode.hasAttributes
   theAttributes = theNode.getAttributes;
   numAttributes = theAttributes.getLength;
   allocCell = cell(1, numAttributes);
   attributes = struct('Name', allocCell, 'Value', ...
                       allocCell);

   for count = 1:numAttributes
```

```
        attrib = theAttributes.item(count-1);
        attributes(count).Name = char(attrib.getName);
        attributes(count).Value = char(attrib.getValue);
    end
end
```

See Also

xmlwrite | xslt

How To

- “What Is an XML Document Object Model (DOM)?”
- “Example — Finding Text in an XML File”

Related Links

- DOM Package Summary (methods and properties for nodes)

Purpose

Write XML Document Object Model node

Syntax

```
xmlwrite(filename,DOMnode)
str = xmlwrite(DOMnode)
```

Description

`xmlwrite(filename,DOMnode)` writes the Document Object Model (DOM) node `DOMnode` to the file `filename`.

`str = xmlwrite(DOMnode)` serializes the DOM node to a string.

Input Arguments

filename

String enclosed in single quotation marks that specifies the name of a local file or a URL.

DOMnode

Document Object Model node, as defined by the World Wide Web consortium. For more information, see "What Is an XML Document Object Model (DOM)?"

Output Arguments

str

String that contains the serialized DOM node as it appears in an XML file.

Examples

Create and view an XML document:

```
docNode = com.mathworks.xml.XMLUtils.createDocument...
    ('root_element')
docRootNode = docNode.getDocumentElement;
docRootNode.setAttribute('attr_name','attr_value');
for i=1:20
    thisElement = docNode.createElement('child_node');
    thisElement.appendChild...
        (docNode.createTextNode(sprintf('%i',i)));
    docRootNode.appendChild(thisElement);
end
```

```
docNode.appendChild(docNode.createComment('this is a comment'));

xmlFileName = [tempname,'.xml'];
xmlwrite(xmlFileName,docNode);
type(xmlFileName);
```

See Also

`xmlread` | `xslt`

How To

- "What Is an XML Document Object Model (DOM)?"
- "Creating an XML File"

Related Links

- DOM Package Summary (methods and properties for nodes)

Purpose

Logical exclusive-OR

Syntax

```
C = xor(A, B)
```

Description

`C = xor(A, B)` performs an exclusive OR operation on the corresponding elements of arrays `A` and `B`. The resulting element `C(i,j,...)` is logical true (1) if `A(i,j,...)` or `B(i,j,...)`, but not both, is nonzero.

A	B	C
Zero	Zero	0
Zero	Nonzero	1
Nonzero	Zero	1
Nonzero	Nonzero	0

Examples

Given `A = [0 0 pi eps]` and `B = [0 -2.4 0 1]`, then

```
C = xor(A,B)
C =
   0   1   1   0
```

To see where either `A` or `B` has a nonzero element and the other matrix does not,

```
spy(xor(A,B))
```

See Also

`all` | `any` | `find` | Elementwise Logical Operators | Short-Circuit Logical Operators

Purpose

Transform XML document using XSLT engine

Syntax

```
result = xslt(source, style, dest)
[result,style] = xslt( ___ )
xslt( ___ ,'-web')
```

Description

`result = xslt(source, style, dest)` transforms an XML document using a stylesheet and returns the resulting document's URL. The function uses these inputs, the first of which is required:

- `source` is the filename or URL of the source XML file. `source` can also specify a DOM node.
- `style` is the filename or URL of an XSL stylesheet.
- `dest` is the filename or URL of the desired output document. If `dest` is absent or empty, the function uses a temporary filename. If `dest` is `'-tostring'`, the function returns the output document as a MATLAB string.

`[result,style] = xslt( ___ )` returns a processed stylesheet appropriate for passing to subsequent XSLT calls as `style`. This prevents costly repeated processing of the stylesheet.

`xslt( ___ ,'-web')` displays the resulting document in the Help Browser.

Tips

MATLAB uses the Saxon XSLT processor, version 6.5.5, which supports XSLT 1.0 expressions. For more information, see `http://saxon.sourceforge.net/saxon6.5.5/`

For additional information on writing XSL stylesheets, see the World Wide Web Consortium (W3C®) web site, `http://www.w3.org/Style/XSL/`.

Examples

This example converts the file `info.xml` using the stylesheet `info.xsl`, writing the output to the file `info.html`. It launches the resulting HTML file in the MATLAB Web Browser.

```
xslt('info.xml', 'info.xsl', 'info.html', '-web')
```

See Also

xmlread | xmlwrite

zeros

Purpose

Create array of all zeros

Syntax

```
X = zeros
X = zeros(n)
X = zeros(sz1,...,szN)
X = zeros(sz)

X = zeros(classname)
X = zeros(n,classname)
X = zeros(sz1,...,szN,classname)
X = zeros(sz,classname)

X = zeros('like',p)
X = zeros(n,'like',p)
X = zeros(sz1,...,szN,'like',p)
X = zeros(sz,'like',p)
```

Description

`X = zeros` returns the scalar `0`.

`X = zeros(n)` returns an `n`-by-`n` matrix of zeros.

`X = zeros(sz1,...,szN)` returns an `sz1`-by-...-by-`szN` array of zeros where `sz1,...,szN` indicates the size of each dimension. For example, `zeros(2,3)` returns a 2-by-3 array of zeros.

`X = zeros(sz)` returns an array of zeros where the size vector, `sz`, defines `size(X)`. For example, `zeros([2,3])` returns a 2-by-3 array of zeros.

`X = zeros(classname)` returns a scalar `0` where the string, `classname`, specifies the data type. For example, `zeros('int8')` returns a scalar, 8-bit integer `0`.

`X = zeros(n,classname)` returns an `n`-by-`n` array of zeros of data type `classname`.

`X = zeros(sz1,...,szN,classname)` returns an `sz1`-by-...-by-`szN` array of zeros of data type `classname`.

`X = zeros(sz,classname)` returns an array of zeros where the size vector, `sz`, defines `size(X)` and `classname` defines `class(X)`.

`X = zeros('like',p)` returns a scalar `0` with the same data type, sparsity, and complexity (real or complex) as the numeric variable, `p`.

`X = zeros(n,'like',p)` returns an `n`-by-`n` array of zeros like `p`.

`X = zeros(sz1,...,szN,'like',p)` returns an `sz1`-by-...-by-`szN` array of zeros like `p`.

`X = zeros(sz,'like',p)` returns an array of zeros like `p` where the size vector, `sz`, defines `size(X)`.

Input Arguments

n - Size of square matrix

integer value

Size of square matrix, specified as an integer value, defines the output as a square, n-by-n matrix of zeros.

- If `n` is `0`, then `X` is an empty matrix.
- If `n` is negative, then it is treated as `0`.

Data Types

`double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

sz1,...,szN - Size of each dimension

two or more integer values

Size of each dimension, specified as two or more integer values, defines X as a sz1-by...-by-szN array.

- If the size of any dimension is 0, then X is an empty array.
- If the size of any dimension is negative, then it is treated as 0.
- If any trailing dimensions greater than 2 have a size of 1, then the output, X, does not include those dimensions.

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

sz - Output size

row vector of integer values

Output size, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension.

- If the size of any dimension is 0, then X is an empty array.
- If the size of any dimension is negative, then it is treated as 0.
- If any trailing dimensions greater than 2 have a size of 1, then the output, X, does not include those dimensions.

Example: sz = [2,3,4] defines X as a 2-by-3-by-4 array.

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

classname - Output class

'double' (default) | 'single' | 'int8' | 'uint8'

Output class, specified as 'double', 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64', or 'uint64'.

Data Types
char

p - Prototype

numeric variable

Prototype, specified as a numeric variable.

Data Types

`double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

Complex Number Support: Yes

Examples

Square Array of Zeros

Create a 4-by-4 array of zeros.

```
X = zeros(4)
```

```
X =

     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
```

3-D Array of Zeros

Create a 2-by-3-by-4 array of zeros.

```
X = zeros(2,3,4);

size(X)
```

```
ans =

     2     3     4
```

Size Defined by Existing Array

Define a 3-by-2 array A.

```
A = [1 4 ; 2 5 ; 3 6];
```

```
sz = size(A)

sz =

     3     2
```

Create an array of zeros that is the same size as A

```
X = zeros(sz)

X =

     0     0
     0     0
     0     0
```

Nondefault Numeric Data Type

Create a 1-by-3 vector of zeros whose elements are 32-bit unsigned integers.

```
X = zeros(1,3,'uint32'),
class(X)

X =

           0            0            0


ans =

uint32
```

Complex Zero

Create a scalar 0 that is not real valued, but instead is complex like an existing array.

Define a complex vector.

```
p = [1+2i 3i];
```

Create a scalar 0 that is complex like p.

```
X = zeros('like',p)
```

```
X =

   0.0000 + 0.0000i
```

Sparse Array

Define a 10-by-10 sparse matrix.

```
p = sparse(10,10,pi);
```

Create a 2-by-3 matrix of zeros that is sparse like p.

```
X = zeros(2,3,'like',p)
```

```
X =

   All zero sparse: 2-by-3
```

Size and Numeric Data Type Defined by Existing Array

Define a 2-by-3 array of 8-bit unsigned integers.

```
p = uint8([1 3 5 ; 2 4 6]);
```

Create an array of zeros that is the same size and data type as p.

```
X = zeros(size(p),'like',p),
class(X)
```

```
X =

    0    0    0
    0    0    0
```

```
ans =

uint8
```

See Also

eye | ones | rand | randn | complex | false | size

Concepts

- “Preallocating Arrays”

Purpose

Compress files into zip file

Syntax

```
zip(zipfile,files)
zip(zipfile,files,rootfolder)
entrynames = zip(zipfile,files,rootfolder)
```

Description

`zip(zipfile,files)` creates a zip file with the name `zipfile` from the list of files and folders specified in `files`. Folders recursively include all of their content.

`zip(zipfile,files,rootfolder)` specifies the path for `files` relative to `rootfolder` instead of the current folder.

`entrynames = zip(zipfile,files,rootfolder)` returns a string cell array of the names of the files contained in `zipfile`. Specifying `rootfolder` is optional.

Input Arguments

zipfile

String that specifies the name of the zip file. If `zipfile` has no extension, MATLAB appends the `.zip` extension.

If `files` includes relative paths, the zip file also contains relative paths. The zip file does not include absolute paths.

files

String or cell array of strings containing the list of files or folders to include in `zipfile`.

Individual files that are on the MATLAB path can be specified as partial path names. Otherwise an individual file can be specified relative to the current folder or with an absolute path.

Folders must be specified relative to the current folder or with absolute paths. On UNIX systems, folders can also start with `~/` or `~`*`username`*`/`, which expands to the current user's home folder or the specified user's home folder, respectively. The wildcard character `*` can be used when specifying files or folders, except when relying on the MATLAB path to resolve a file name or partial path name.

rootfolder

String that specifies the root of the paths for the files to zip.

Relative paths in the zip file reflect the relative paths in `files`, and do not include path information from `rootfolder`.

Default: current folder (`'.'`)

Output Arguments

entrynames

Cell array of strings that contain the paths to the files in `zipfile`. If `files` includes relative paths, `entrynames` also contains relative paths.

Examples

Zip a File

Create a zip file of the file `membrane.m`, which is in the MATLAB `general` folder. Save the zip file `tmwlogo.zip` in the current folder:

```
file = fullfile(matlabroot,'toolbox','matlab','general','membrane.m');
zip('tmwlogo',file);
```

Zip Selected Files

Suppose that your system has a folder named `d:/myfiles`. Zip the files `membrane.m` and `logo.m`, which are on the MATLAB search path, into a file named `tmwlogo.zip`:

```
myfile = fullfile('d:','myfiles','tmwlogo.zip');
zip(myfile,{'membrane.m','logo.m'});
```

Zip all `.m` and `.mat` files in the current folder to the file `backup.zip`:

```
zip('backup',{'*.m','*.mat'});
```

Zip a Folder

Suppose that your current folder contains a subfolder named `mywork`. Zip the contents of all subfolders of `mywork`, and store the relative paths in the zip file:

```
zip('myfiles.zip','mywork');
```

Zip Between Folders

Suppose that you have files `thesis.doc` and `defense.ppt` in `d:/PhD`. Zip these files into `thesis.zip`, one level up from the current folder:

```
zip('../thesis.zip',{'thesis.doc','defense.ppt'},'d:/PhD');
```

Alternatives

To zip files in the Current Folder browser, select the files, right-click to open the context menu, and then select **Create Zip File**.

See Also

`gzip` | `gunzip` | `tar` | `untar` | `unzip`

zoom

Purpose

Turn zooming on or off or magnify by factor

Syntax

```
zoom on
zoom off
zoom out
zoom reset
zoom
zoom xon
zoom yon
zoom(factor)
zoom(fig, option)
h = zoom(figure_handle)
```

Description

`zoom on` turns on interactive zooming. When interactive zooming is enabled in a figure, pressing a mouse button while your cursor is within an axes zooms into the point or out from the point beneath the mouse. Zooming changes the axes limits. When using zoom mode, you

- Zoom in by positioning the mouse cursor where you want the center of the plot to be and either
  - Press the mouse button or
  - Rotate the mouse scroll wheel away from you (upward).
- Zoom out by positioning the mouse cursor where you want the center of the plot to be and either
  - Simultaneously press **Shift** and the mouse button, or
  - Rotate the mouse scroll wheel toward you (downward).

Each mouse click or scroll wheel click zooms in or out by a factor of 2.

Clicking and dragging over an axes when zooming in is enabled draws a rubberband box. When you release the mouse button, the axes zoom in to the region enclosed by the rubberband box.

Double-clicking over an axes returns the axes to its initial zoom setting in both zoom-in and zoom-out modes.

`zoom off` turns interactive zooming off.

`zoom out` returns the plot to its initial zoom setting.

`zoom reset` remembers the current zoom setting as the initial zoom setting. Later calls to `zoom out`, or double-clicks when interactive `zoom` mode is enabled, will return to this zoom level.

`zoom` toggles the interactive zoom status between off and on (restoring the most recently used zoom tool).

`zoom xon` and `zoom yon` set `zoom on` for the *x*- and *y*-axis, respectively.

`zoom(factor)` zooms in or out by the specified zoom factor, without affecting the interactive zoom mode. Values greater than 1 zoom in by that amount, while numbers greater than 0 and less than 1 zoom out by `1/factor`.

`zoom(fig, option)` Any of the preceding options can be specified on a figure other than the current figure using this syntax.

`h = zoom(figure_handle)` returns a zoom *mode object* for the figure `figure_handle` for you to customize the mode's behavior.

Using Zoom Mode Objects

Access the following properties of zoom mode objects via `get` and modify some of them using `set`.

- *`Enable`* `'on'|'off'` — Specifies whether this figure mode is currently enabled on the figure
- `FigureHandle <handle>` — The associated figure handle, a read-only property that cannot be set
- *`Motion`* `'horizontal'|'vertical'|'both'` — The type of zooming enabled for the figure
- *`Direction`* `'in'|'out'` — The direction of the zoom operation
- *`RightClickAction`* `'InverseZoom'|'PostContextMenu'` — The behavior of a right-click action

  A value of `'InverseZoom'` causes a right-click to zoom out. A value of `'PostContextMenu'` displays a context menu. This setting persists between MATLAB sessions.

- `UIContextMenu <handle>` — Specifies a custom context menu to be displayed during a right-click action

  This property is ignored if the `RightClickAction` property has been set to `'on'`.

Zoom Mode Callbacks

You can program the following callbacks for zoom mode operations.

- `ButtonDownFilter <function_handle>` — Function to intercept `ButtonDown` events

  The application can inhibit the zoom operation under circumstances the programmer defines, depending on what the callback returns. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks), as follows:

  ```
  function [res] = myfunction(obj,event_obj)
  % obj          handle to the object that has been clicked on
  % event_obj    struct for event data (empty in this release)
  % res [output] a logical flag determines whether the zoom
  %              operation should take place(for 'res' set to 'false'
  %              or the 'ButtonDownFcn' property of the object should
  %              take precedence (when 'res' is 'true')
  ```

- `ActionPreCallback <function_handle>` — Function to execute before zooming

  Set this callback if you want to execute code when a zoom operation starts. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks), as follows:

  ```
  function myfunction(obj,event_obj)
  % obj         handle to the figure that has been clicked on.
  % event_obj   object containing struct of event data
  ```

  The event data has the following field.

Axes	The handle of the axes that is being zoomed

- ActionPostCallback <function_handle> — Function to execute after zooming

  Set this callback if you want to execute code when a zoom operation finishes. The input function handle should reference a function with two implicit arguments (similar to Handle Graphics object callbacks), as follows:

```
function myfunction(obj,event_obj)
% obj         handle to the figure that has been clicked on
% event_obj   object containing struct of event data (same as the
%             event data of the 'ActionPreCallback' callback)
```

Zoom Mode Utility Functions

The following functions in zoom mode query and set certain of its properties.

- flags = isAllowAxesZoom(h,axes) — Function querying permission to zoom axes

  Calling the function isAllowAxesZoom on the zoom object, h, with a vector of axes handles, axes, as input returns a logical array of the same dimension as the axes handle vector, which indicates whether a zoom operation is permitted on the axes objects.

- setAllowAxesZoom(h,axes,flag) — Function to set permission to zoom axes

  Calling the function setAllowAxesZoom on the zoom object, h, with a vector of axes handles, axes, and a logical scalar, flag, either allows or disallows a zoom operation on the axes objects.

- *info* = getAxesZoomMotion(h,axes) — Function to get style of zoom operations

Calling the function `getAxesZoomMotion` on the zoom object, `H`, with a vector of axes handles, `axes`, as input returns a character cell array of the same dimension as the axes handle vector, which indicates the type of zoom operation for each axes. Possible values for the type of operation are `'horizontal'`, `'vertical'`, or `'both'`.

- `setAxesZoomMotion(h,axes,style)` — Function to set style of zoom operations

  Calling the function `setAxesZoomMotion` on the zoom object, `h`, with a vector of axes handles, `axes`, and a character array, `style`, sets the style of zooming on each axes.

Examples

Example 1 — Entering Zoom Mode

Plot a graph and turn on Zoom mode:

```
plot(1:10);
zoom on
% zoom in on the plot
```

Example 2 — Constrained Zoom

Create zoom mode object and constrain to *x*-axis zooming:

```
plot(1:10);
 h = zoom;
set(h,'Motion','horizontal','Enable','on');
% zoom in on the plot in the horizontal direction.
```

Example 3 — Constrained Zoom in Subplots

Create four axes as subplots and set zoom style differently for each by setting a different property for each axes handle:

```
ax1 = subplot(2,2,1);
plot(1:10);
h = zoom;
ax2 = subplot(2,2,2);
plot(rand(3));
```

```
setAllowAxesZoom(h,ax2,false);
ax3 = subplot(2,2,3);
plot(peaks);
setAxesZoomMotion(h,ax3,'horizontal');
ax4 = subplot(2,2,4);
contour(peaks);
setAxesZoomMotion(h,ax4,'vertical');
% Zoom in on the plots.
```

Example 4 — Coding a ButtonDown Callback

Create a buttonDown callback for zoom mode objects to trigger. Copy the following code to a new file, execute it, and observe zooming behavior:

```
function demo
% Allow a line to have its own 'ButtonDownFcn' callback.
hLine = plot(rand(1,10));
set(hLine,'ButtonDownFcn','disp(''This executes'')');
set(hLine,'Tag','DoNotIgnore');
h = zoom;
set(h,'ButtonDownFilter',@mycallback);
set(h,'Enable','on');
% mouse click on the line
%
function [flag] = mycallback(obj,event_obj)
% If the tag of the object is 'DoNotIgnore', then return true.
objTag = get(obj,'Tag');
if strcmpi(objTag,'DoNotIgnore')
   flag = true;
else
   flag = false;
end
```

Example 5 — Coding Pre- and Post-Callback Behavior

Create callbacks for pre- and post-buttonDown events for zoom mode objects to trigger. Copy the following code to a new file, execute it, and observe zoom behavior:

```
function demo
% Listen to zoom events
plot(1:10);
h = zoom;
set(h,'ActionPreCallback',@myprecallback);
set(h,'ActionPostCallback',@mypostcallback);
set(h,'Enable','on');
%
function myprecallback(obj,evd)
disp('A zoom is about to occur.');
%
function mypostcallback(obj,evd)
newLim = get(evd.Axes,'XLim');
msgbox(sprintf('The new X-Limits are [%.2f %.2f].',newLim));
```

Example 6 — Creating a Context Menu for Zoom Mode

Coding a context menu that lets the user to switch to Pan mode by right-clicking:

```
figure;plot(magic(10))
hCMZ = uicontextmenu;
hZMenu = uimenu('Parent',hCMZ,'Label','Switch to pan','Callback','pan(gcb
hZoom = zoom(gcf);
set(hZoom,'UIContextMenu',hCMZ);
zoom('on')
```

You cannot add items to the built-in zoom context menu, but you can replace it with your own.

Tips

`zoom` changes the axes limits by a factor of 2 (in or out) each time you press the mouse button while the cursor is within an axes. You can

also click and drag the mouse to define a zoom area, or double-click to return to the initial zoom level.

You can create a zoom mode object once and use it to customize the behavior of different axes, as Example 3 illustrates. You can also change its callback functions on the fly.

Note Do not change figure callbacks within an interactive mode. While a mode is active (when panning, zooming, etc.), you will receive a warning if you attempt to change any of the figure's callbacks and the operation will not succeed. The one exception to this rule is the figure `WindowButtonMotionFcn` callback, which can be changed from within a mode. Therefore, if you are creating a GUI that updates a figure's callbacks, the GUI should somehow keep track of which interactive mode is active, if any, before attempting to do this.

When you assign different zoom behaviors to different `subplot` axes via a mode object and then link them using the `linkaxes` function, the behavior of the axes you manipulate with the mouse carries over to the linked axes, regardless of the behavior you previously set for the other axes.

Alternatives

Use the **Zoom** tools on the figure toolbar to zoom in or zoom out on a plot, or select **Zoom In** or **Zoom Out** from the figure's **Tools** menu. For details, see "Zooming in Graphs".

See Also

`linkaxes` | `pan` | `rotate3d`

Index

Symbols and Numerics

& 1-71 1-78
' 1-60
* 1-60
+ 1-60
- 1-60
/ 1-60
: 1-85
< 1-69
> 1-69
@ 1-1869
\ 1-60
^ 1-60
| 1-71 1-78
~ 1-71 1-78
&& 1-78
== 1-69
]) 1-83
|| 1-78
~= 1-69
1-norm 1-4206
2-norm (estimate of) 1-3608

A

abs 1-88
absolute accuracy
  BVP 1-587
  DDE 1-1269
  ODE 1-3751
absolute value 1-88
Accelerator
  Uimenu property 1-5792
accumarray 1-89
accuracy
  of linear equation solution 1-1019
  of matrix inversion 1-1019
acos 1-96
acosh 1-99
acot 1-100
acoth 1-103
acsc 1-104
acsch 1-107
activelegend 1-3966
actxcontrol 1-108
Adams-Bashforth-Moulton ODE solver 1-3656 1-3671 1-3686 1-3701 1-3716 1-3731 1-3746
addCause, MException method 1-134
addevent 1-137
addframe
  AVI files 1-157
addition (arithmetic operator) 1-60
addlistener 1-159
addpref function 1-171
addprop dynamicprops method 1-173
addressing selected array elements 1-85
addsampletocollection 1-178
addtodate 1-180
addts 1-182
adjacency graph 1-1396
align function 1-194
aligning scattered data
  multi-dimensional 1-3454
ALim, Axes property 1-344
all 1-200
allchild function 1-202
AlphaData
  image property 1-2611
  surface property 1-5121
  surfaceplot property 1-5145
AlphaDataMapping
  image property 1-2612
  patch property 1-3855
  surface property 1-5122
  surfaceplot property 1-5145
AmbientLightColor, Axes property 1-344
AmbientStrength
  Patch property 1-3856
  Surface property 1-5122
  surfaceplot property 1-5146

amd 1-210
analytical partial derivatives (BVP) 1-589
and 1-214
and (function equivalent for &) 1-75
angle 1-216
annotating graphs
  in plot edit mode 1-3967
Annotation
  areaseries property 1-272
  contourgroup property 1-1047
  errorbarseries property 1-1488
  hggroup property 1-2498
  hgtransform property 1-2526
  image property 1-2612
  line property 1-438 1-3041
  lineseries property 1-3059
  Patch property 1-3856
  quivergroup property 1-4147
  rectangle property 1-4380
  scattergroup property 1-4633
  stairseries property 1-4878
  stemseries property 1-4946
  Surface property 1-5123
  surfaceplot property 1-5146
  text property 1-5521
annotation function 1-217
ans 1-262
anti-diagonal 1-2432
any 1-263
arccosecant 1-104
arccosine 1-96
arccotangent 1-100
arcsecant 1-295
arctangent 1-308
  four-quadrant 1-309
arguments
  checking number of inputs 1-3437
  checking number of outputs 1-3446
  passing variable numbers of 1-6019
arithmetic operations, matrix and array
  distinguished 1-60
arithmetic operators
  reference 1-60
array
  addressing selected elements of 1-85
  dimension
    rearrange 1-1777
  flip dimension of 1-1777
  left division (arithmetic operator) 1-62
  maximum elements of 1-3232
  mean elements of 1-3237
  minimum elements of 1-3344
  multiplication (arithmetic operator) 1-61
  power (arithmetic operator) 1-62
  rearrange
    dimension 1-1777
  removing first n singleton dimensions
    of 1-4745
  removing singleton dimensions of 1-4848
  reshaping 1-4501
  reverse dimension of 1-1777
  right division (arithmetic operator) 1-61
  shift circularly 1-915
  shifting dimensions of 1-4745
  size of 1-4758
  sorting elements of 1-4780
  structure 1-1981 1-4716
  sum of elements 1-5100
  swapping dimensions of 1-2812 1-3935
  transpose (arithmetic operator) 1-62
arrays
  detecting empty 1-2828
arrays, structure
  field names of 1-1640
arrowhead matrix 1-1001
ASCII
  delimited files
    writing 1-1391
ASCII data

converting sparse matrix after loading from 1-4797
reading 1-1387
reading from disk 1-3116
saving to disk 1-4576

`asec` 1-295
`asech` 1-298
`asinh` 1-303
aspect ratio of axes 1-1160 1-3891
`assert` 1-304
`assignin` 1-306
`atan` 1-308
`atan2` 1-309
`atanh` 1-315
`.au` files
  reading 1-328
  writing 1-330
audio
  saving in AVI format 1-331
  signal conversion 1-3034 1-3421
`audiodevinfo` 1-316
`audioplayer` 1-318
`audiorecorder` 1-322
`aufinfo` 1-327
`auread` 1-328
`AutoScale`
  quivergroup property 1-4148
`AutoScaleFactor`
  quivergroup property 1-4148
autoselection of OpenGL 1-1680
`auwrite` 1-330
average of array elements 1-3237
average,running 1-1732
`avi` 1-331
`avifile` 1-331
`aviinfo` 1-335
`aviread` 1-337
`axes` 1-338
  editing 1-3967
  setting and querying data aspect ratio 1-1160
  setting and querying limits 1-6198 1-6202 1-6206
  setting and querying plot box aspect ratio 1-3891
Axes
  creating 1-338
  defining default properties 1-339
  fixed-width font 1-360
  property descriptions 1-343
`axis` 1-382
azimuth (spherical coordinates) 1-4813
azimuth of viewpoint 1-6054

B

`BackFaceLighting`
  Surface property 1-5124
  surfaceplot property 1-5148
`BackFaceLighting`patch property 1-3858
`BackgroundColor`
  annotation textbox property 1-252
  Text property 1-5522
  Uitable property 1-5878
`BackGroundColor`
  Uicontrol property 1-5738
badly conditioned 1-4206
`balance` 1-388
`BarLayout`
  barseries property 1-439
`BarWidth`
  barseries property 1-439
base to decimal conversion 1-459
base two operations
  conversion from decimal to binary 1-1283
  logarithm 1-3136
`base2dec` 1-459
`BaseLine`
  barseries property 1-439
  stem property 1-4947
`BaseValue`

areaseries property 1-273
barseries property 1-440
stem property 1-4947
beep 1-460
`BeingDeleted`
areaseries property 1-273
barseries property 1-441
contour property 1-1048
errorbar property 1-1489
group property 1-1645 1-2613 1-5524
hggroup property 1-2499
hgtransform property 1-2527
light property 1-3023
line property 1-3042
lineseries property 1-3060
quivergroup property 1-4148
rectangle property 1-4381
scatter property 1-4634
stairseries property 1-4879
stem property 1-4947
surface property 1-5124
surfaceplot property 1-5148
transform property 1-3858
Uipushtool property 1-5840
Uitable property 1-5879
Uitoggletool property 1-5909
Uitoolbar property 1-5923
bench 1-462
benchmark 1-462
Bessel functions
first kind 1-471
modified, first kind 1-468
modified, second kind 1-475
second kind 1-479
Bessel's equation
(defined) 1-471
modified (defined) 1-468
`besseli` 1-468
`besselj` 1-471
`besselk` 1-475
`bessely` 1-479
`beta` 1-483
beta function
(defined) 1-483
incomplete (defined) 1-485
natural logarithm 1-488
`betainc` 1-485
`betaln` 1-488
`bicg` 1-489
`bicgstab` 1-497
`bicgstabl` 1-504
BiConjugate Gradients method 1-489
BiConjugate Gradients Stabilized method 1-497 1-504
`bin2dec` 1-510
binary data
reading from disk 1-3116
saving to disk 1-4576
binary to decimal conversion 1-510
bisection search 1-1899
bit depth
querying 1-2639
bitmaps
writing 1-2671
`bitmax` 1-521
`blanks` 1-538
removing trailing 1-1280
`blkdiag` 1-539
BMP files
writing 1-2671
bold font
TeX characters 1-5548
boundary value problems 1-594
`box` 1-540
`Box`, Axes property 1-345
braces, curly (special characters) 1-81
brackets (special characters) 1-81
`break` 1-541
breakpoints
listing 1-1232

removing 1-1219
resuming execution from 1-1222
`brush` 1-546
`bsxfun` 1-557
Buckminster Fuller 1-5203
`BusyAction`
areaseries property 1-273
Axes property 1-345
barseries property 1-441
contour property 1-1048
errorbar property 1-1490
Figure property 1-1646
hggroup property 1-2500
hgtransform property 1-2528
Image property 1-2614
Light property 1-3023
line property 1-3043
Line property 1-3060
patch property 1-3858
quivergroup property 1-4149
rectangle property 1-4382
Root property 1-4539
scatter property 1-4634
stairseries property 1-4879
stem property 1-4948
Surface property 1-5124
surfaceplot property 1-5148
Text property 1-5524
Uicontextmenu property 1-5717
Uicontrol property 1-5739
Uimenu property 1-5794
Uipushtool property 1-5841
Uitable property 1-5879
Uitoggletool property 1-5910
Uitoolbar property 1-5923
`ButtonDownFcn`
area series property 1-274
Axes property 1-346
barseries property 1-442
contour property 1-1049
errorbar property 1-1490
Figure property 1-1646
hggroup property 1-2500
hgtransform property 1-2528
Image property 1-2614
Light property 1-3024
Line property 1-3043
lineseries property 1-3061
patch property 1-3859
quivergroup property 1-4149
rectangle property 1-4382
Root property 1-4539
scatter property 1-4635
stairseries property 1-4880
stem property 1-4948
Surface property 1-5125
surfaceplot property 1-5149
Text property 1-5525
Uicontrol property 1-5740
Uitable property 1-5880
BVP solver properties
analytical partial derivatives 1-589
error tolerance 1-587
Jacobian matrix 1-589
mesh 1-591
singular BVPs 1-590
solution statistics 1-592
vectorization 1-588
`bvp4c` 1-566
`bvp5c` 1-577
`bvpget` 1-582
`bvpinit` 1-583
`bvpset` 1-586
`bvpxtend` 1-594

C

`calendar` 1-595
call history 1-4061
`CallBack`

Uicontextmenu property 1-5718
Uicontrol property 1-5741
Uimenu property 1-5794
CallbackObject, Root property 1-4539
camdolly 1-601
camera
dollying position 1-601
moving camera and target postions 1-601
positioning to view objects 1-607
rotating around camera target 1-609 1-611
rotating around viewing axis 1-617
setting and querying position 1-613
setting and querying projection type 1-615
setting and querying target 1-618
setting and querying up vector 1-620
setting and querying view angle 1-622
CameraPosition, Axes property 1-347
CameraPositionMode, Axes property 1-348
CameraTarget, Axes property 1-348
CameraTargetMode, Axes property 1-348
CameraUpVector, Axes property 1-348
CameraUpVectorMode, Axes property 1-349
CameraViewAngle, Axes property 1-349
CameraViewAngleMode, Axes property 1-349
camlookat 1-607
camorbit 1-609
campan 1-611
campos 1-613
camproj 1-615
camroll 1-617
camtarget 1-618
camup 1-620
camva 1-622
camzoom 1-624
cart2pol 1-628
cart2sph 1-630
Cartesian coordinates 1-628 1-630 1-3987 1-4813
case
lower to upper 1-5983
upper to lower 1-3148
cast 1-634
cat 1-636
caxis 1-640
Cayley-Hamilton theorem 1-4007
cd 1-646
CData
Image property 1-2615
scatter property 1-4636
Surface property 1-5126
surfaceplot property 1-5150
Uicontrol property 1-5741
Uipushtool property 1-5841
Uitoggletool property 1-5910
CDataMapping
Image property 1-2617
patch property 1-3861
Surface property 1-5127
surfaceplot property 1-5150
CDataMode
surfaceplot property 1-5151
CDatapatch property 1-3859
CDataSource
scatter property 1-4636
surfaceplot property 1-5151
cdf2rdf 1-653
cdfepoch 1-655
cdfinfo 1-657
cdflib
summary of capabilities 1-661
cdfread 1-849
cdfwrite 1-853
ceil 1-856
cell array
structure of, displaying 1-877
cell2struct 1-862
celldisp 1-871
CellEditCallback
Uitable property 1-5881
cellplot 1-877
CellSelectionCallback

Uitable property 1-5883
`cgs` 1-880
`char` 1-887
characters
conversion, in serial format specification string 1-1826
check boxes 1-5725
`Checked`, Uimenu property 1-5795
checkerboard pattern (example) 1-4493
`checkin` 1-894
examples 1-895
options 1-894
`checkout` 1-896
examples 1-897
options 1-896
child functions 1-4056
`Children`
areaseries property 1-275
Axes property 1-351
barseries property 1-442
contour property 1-1050
errorbar property 1-1491
Figure property 1-1647
hggroup property 1-2501
hgtransform property 1-2529
Image property 1-2618
Light property 1-3024
Line property 1-3044
lineseries property 1-3061
patch property 1-3862
quivergroup property 1-4150
rectangle property 1-4383
Root property 1-4539
scatter property 1-4636
stairseries property 1-4881
stem property 1-4949
Surface property 1-5127
surfaceplot property 1-5152
Text property 1-5526
Uicontextmenu property 1-5718
Uicontrol property 1-5742
Uimenu property 1-5796
Uitable property 1-5883
Uitoolbar property 1-5924
`chol` 1-899
Cholesky factorization 1-899
lower triangular factor 1-3834
preordering for 1-1001
`cholinc` 1-904
`cholupdate` 1-912
circle
`rectangle` function 1-4375
`circshift` 1-915
`cla` 1-919
classes
field names 1-1640
loaded 1-2703
`clc` 1-937 1-946 1-4744
`clear`
serial port I/O 1-945
clearing
Command Window 1-937
Java import list 1-940
`clf` 1-946
`ClickedCallback`
Uipushtool property 1-5841
Uitoggletool property 1-5911
`CLim`, Axes property 1-351
`CLimMode`, Axes property 1-352
`clipboard` 1-947
`Clipping`
areaseries property 1-275
Axes property 1-352
barseries property 1-443
contour property 1-1050
errrobar property 1-1491
Figure property 1-1648
hggroup property 1-2501
hgtransform property 1-2529
Image property 1-2618

Light property 1-3024
Line property 1-3045
lineseries property 1-3062
quivergroup property 1-4150
rectangle property 1-4383
Root property 1-4540
scatter property 1-4637
stairseries property 1-4881
stem property 1-4949
Surface property 1-5128
surfaceplot property 1-5152
Text property 1-5526
Uicontrol property 1-5742
Uitable property 1-5883
`Clipping`patch property 1-3862
`close` 1-949
AVI files 1-952
`CloseRequestFcn`, Figure property 1-1648
closest point search 1-1414
closest triangle search 1-5682
closing
MATLAB 1-4138
cmapeditor 1-981
`cmpermute` 1-959
`cmunique` 1-960
Code Analyzer
function for entire folder 1-3374
HTML report 1-3374
`colamd` 1-963
colon operator 1-85
`Color`
annotation arrow property 1-223
annotation doublearrow property 1-227
annotation line property 1-235
annotation textbox property 1-252
Axes property 1-352
errorbar property 1-1491
Figure property 1-1650
Light property 1-3024
Line property 1-3045
lineseries property 1-3062
quivergroup property 1-4150
stairseries property 1-4881
stem property 1-4950
Text property 1-5526
textarrow property 1-241
color of fonts, see also `FontColor` property 1-5548
`colorbar` 1-967
`colormap` 1-975
editor 1-981
`Colormap`, Figure property 1-1650
colormaps
converting from RGB to HSV 1-4516
plotting RGB components 1-4521
rearranging colors in 1-959
removing duplicate entries in 1-960
`ColorOrder`, Axes property 1-352
`ColorSpec` 1-999
`colperm` 1-1001
`ColumnEditable`
Uitable property 1-5883
`ColumnFormat`
Uitable property 1-5884
`ColumnName`
Uitable property 1-5890
`ColumnWidth`
Uitable property 1-5890
COM
object methods
`actxcontrol` 1-108
`delete` 1-1332
`events` 1-1527
`get` 1-1957
`inspect` 1-2723
`load` 1-3121
`move` 1-3398
`propedit` 1-4065
`save` 1-4583
`set` 1-4690
server methods

`Execute` 1-1529
`Feval` 1-1610
`comet` 1-1004
`comet3` 1-1006
comma (special characters) 1-82
command syntax 1-5206
Command Window
clearing 1-937
cursor position 1-2562
get width 1-1008
`commandhistory` 1-1007
commands
help for 1-2464
`commandwindow` 1-1008
comments
block of 1-83
`compan` 1-1009
companion matrix 1-1009
`compass` 1-1010
`CompilerConfiguration` 1-3329
`CompilerConfigurationDetails` 1-3329
complementary error function
(defined) 1-1476
complete elliptic integral
(defined) 1-1441
modulus of 1-1439 1-1441
`complex` 1-1012 1-2602
exponential (defined) 1-1537
logarithm 1-3133 to 1-3134
numbers 1-2571
numbers, sorting 1-4780 1-4784
phase angle 1-216
See also imaginary
complex conjugate 1-1028
sorting pairs of 1-1112
complex data
creating 1-1012
complex numbers, magnitude 1-88
complex Schur form 1-4652
compression
lossy 1-2676
concatenation
of arrays 1-636
`cond` 1-1019
`condeig` 1-1020
`condest` 1-1021
condition number of matrix 1-1019 1-4206
improving 1-388
`coneplot` 1-1023
`conj` 1-1028
conjugate, complex 1-1028
sorting pairs of 1-1112
context menu 1-5713
continuation (`...`, special characters) 1-82
continue 1-1029
continued fraction expansion 1-4201
contour
and mesh plot 1-1563
filled plot 1-1555
functions 1-1551
of mathematical expression 1-1552
with surface plot 1-1586
`contour3` 1-1039
`contourc` 1-1042
`contourf` 1-1044
`ContourMatrix`
contour property 1-1050
contours
in slice planes 1-1069
`contourslice` 1-1069
`contrast` 1-1072
`conv` 1-1073
`conv2` 1-1075
conversion
base to decimal 1-459
binary to decimal 1-510
Cartesian to cylindrical 1-628
Cartesian to polar 1-628
complex diagonal to real block diagonal 1-653
cylindrical to Cartesian 1-3987

decimal number to base 1-1277 1-1282
decimal to binary 1-1283
decimal to hexadecimal 1-1284
full to sparse 1-4794
hexadecimal to decimal 1-2492
integer to string 1-2736
lowercase to uppercase 1-5983
matrix to string 1-3196
numeric array to logical array 1-3137
numeric array to string 1-3625
partial fraction expansion to pole-residue 1-4503
polar to Cartesian 1-3987
pole-residue to partial fraction expansion 1-4503
real to complex Schur form 1-4572
spherical to Cartesian 1-4813
string matrix to cell array 1-879
string to numeric array 1-4974
uppercase to lowercase 1-3148
vector to character string 1-887
conversion characters in serial format specification string 1-1826
convex hulls
multidimensional vizualization 1-1082
two-dimensional visualization 1-1080
`convhull` 1-1080
`convhulln` 1-1082
`convn` 1-1084
convolution 1-1073
inverse. *See* deconvolution
two-dimensional 1-1075
coordinate system and viewpoint 1-6054
coordinates
Cartesian 1-628 1-630 1-3987 1-4813
cylindrical 1-628 1-630 1-3987
polar 1-628 1-630 1-3987
spherical 1-4813
coordinates. 1-628
See also conversion
`copyfile` 1-1094
copying
files and folders 1-1094
`copyobj` 1-1097
`corrcoef` 1-1099
cosecant
hyperbolic 1-1127
inverse 1-104
inverse hyperbolic 1-107
`cosh` 1-1105
cosine
hyperbolic 1-1105
inverse 1-96
inverse hyperbolic 1-99
`cot` 1-1106
cotangent 1-1106
hyperbolic 1-1109
inverse 1-100
inverse hyperbolic 1-103
`coth` 1-1109
`cov` 1-1110
`cplxpair` 1-1112
`cputime` 1-1113
`create`, RandStream method 1-1114
`CreateFcn`
areaseries property 1-275
Axes property 1-353
barseries property 1-443
contour property 1-1051
errorbar property 1-1492
Figure property 1-1651
group property 1-2529
hggroup property 1-2501
Image property 1-2618
Light property 1-3025
Line property 1-3045
lineseries property 1-3062
patch property 1-3862
quivergroup property 1-4151
rectangle property 1-4384

Root property 1-4540
scatter property 1-4637
stairseries property 1-4882
stemseries property 1-4950
Surface property 1-5128
surfaceplot property 1-5152
Text property 1-5526
Uicontextmenu property 1-5718
Uicontrol property 1-5742
Uimenu property 1-5796
Uipushtool property 1-5842
Uitable property 1-5891
Uitoggletool property 1-5911
Uitoolbar property 1-5924
`cross` 1-1123
cross product 1-1123
`csc` 1-1124
`csch` 1-1127
`csvwrite` 1-1133
`ctranspose` (function equivalent for `\q`) 1-66
cubic interpolation 1-2777 1-2781 1-2785 1-3902
piecewise Hermite 1-2769
cubic spline interpolation
one-dimensional 1-2769 1-2777 1-2781 1-2785
`cumtrapz` 1-1145
curl 1-1147
curly braces (special characters) 1-81
current folder 1-646
changing 1-646
See also search path
`CurrentAxes` 1-1652
`CurrentAxes`, Figure property 1-1652
`CurrentCharacter`, Figure property 1-1652
`CurrentFigure`, Root property 1-4540
`CurrentObject`, Figure property 1-1653
`CurrentPoint`
Axes property 1-354
Figure property 1-1653
cursor images
reading 1-2657
cursor position 1-2562
`Curvature`, rectangle property 1-4385
curve fitting (polynomial) 1-3999
`customverctrl` 1-1151
Cuthill-McKee ordering, reverse 1-5193 1-5203
`cylinder` 1-1152
cylindrical coordinates 1-628 1-630 1-3987

D

`daqread` 1-1155
`daspect` 1-1160
data
ASCII
reading from disk 1-3116
ASCII, saving to disk 1-4576
binary, saving to disk 1-4576
computing 2-D stream lines 1-4988
computing 3-D stream lines 1-4990
formatted
reading from files 1-1847
isosurface from volume data 1-2889
reading binary from disk 1-3116
reading from files 1-5554
reducing number of elements in 1-4401
smoothing 3-D 1-4775
`Data`
Uitable property 1-5892
data aspect ratio of axes 1-1160
data brushing
different plot types 1-547
gestures for 1-552
restrictions on 1-549
data types
complex 1-1012
data, aligning scattered
multi-dimensional 1-3454
data, ASCII

converting sparse matrix after loading from 1-4797
`DataAspectRatio`, Axes property 1-356
`DataAspectRatioMode`, Axes property 1-358
`datatipinfo` 1-1175
`date` 1-1176
date and time functions 1-1465
`dbclear` 1-1219
`dbcont` 1-1222
`dbdown` 1-1223
`dblquad` 1-1224
`dbmex` 1-1226
`dbquit` 1-1227
`dbstack` 1-1229
`dbstatus` 1-1232
`dbstep` 1-1234
`dbtype` 1-1245
`dbup` 1-1246
DDE solver properties
error tolerance 1-1268
event location 1-1273
solver output 1-1269
step size 1-1272
`dde23` 1-1247
`ddeget` 1-1251
`ddephas2` output function 1-1271
`ddephas3` output function 1-1271
`ddeplot` output function 1-1271
`ddeprint` output function 1-1271
`ddesd` 1-1262
`ddeset` 1-1267
`deal` 1-1277
`deblank` 1-1280
debugging
changing workspace context 1-1223
changing workspace to calling file 1-1246
displaying function call stack 1-1229
files 1-4056
function 1-2950
MEX-files on UNIX 1-1226
removing breakpoints 1-1219
resuming execution from breakpoint 1-1234
stepping through lines 1-1234
`dec2base` 1-1277 1-1282
`dec2bin` 1-1283
`dec2hex` 1-1284
`decic` function 1-1286
decimal number to base conversion 1-1277 1-1282
decimal point (.)
(special characters) 1-82
to distinguish matrix and array operations 1-60
decomposition
Dulmage-Mendelsohn 1-1395
"economy-size" 1-5182
Schur 1-4652
singular value 1-4200 1-5182
`deconv` 1-1288
deconvolution 1-1288
definite integral 1-4111
del operator 1-1289
`del2` 1-1289
Delaunay tessellation
multidimensional vizualization 1-1302
`delaunayn` 1-1302
`delete` 1-1330 1-1332
serial port I/O 1-1335
timer object 1-1336
`delete` handle method 1-1334
`DeleteFcn`
areaseries property 1-276
Axes property 1-359
barseries property 1-444
contour property 1-1052
errorbar property 1-1492
Figure property 1-1655
hggroup property 1-2502
hgtransform property 1-2530
Image property 1-2619

Light property 1-3026
lineseries property 1-3063
quivergroup property 1-4151
Root property 1-4541
scatter property 1-4638
stairseries property 1-4882
stem property 1-4951
Surface property 1-5128
surfaceplot property 1-5153
Text property 1-5527
Uicontextmenu property 1-5720 1-5743
Uimenu property 1-5797
Uipushtool property 1-5843
Uitable property 1-5893
Uitoggletool property 1-5912
Uitoolbar property 1-5926
`DeleteFcn`, line property 1-3046
`DeleteFcn`, rectangle property 1-4385
`DeleteFcn`patch property 1-3863
deleting
files 1-1330
`delevent` 1-1339
delimiters in ASCII files 1-1387 1-1391
`delsamplefromcollection` 1-1340
density
of sparse matrix 1-3604
`depdir` 1-1343
dependence, linear 1-5092
dependent functions 1-4056
derivative
approximate 1-1362
polynomial 1-3996
desktop
starting without 1-3218
`det` 1-1351
detecting
alphabetic characters 1-2860
empty arrays 1-2828
global variables 1-2846
logical arrays 1-2861
positive, negative, and zero array
elements 1-4752
sparse matrix 1-2910
determinant of a matrix 1-1351
`detrend` 1-1352
`deval` 1-1354
`diag` 1-1356
diagonal 1-1356
anti- 1-2432
`k`-th (illustration) 1-5652
main 1-1356
sparse 1-4799
`dialog` 1-1358
dialog box
error 1-1507
help 1-2469
input 1-2710
list 1-3111
message 1-3415
print 1-4038
question 1-4134
warning 1-6090
`diary` 1-1360
`Diary`, Root property 1-4541
`DiaryFile`, Root property 1-4541
`diff` 1-1362
differences
between adjacent array elements 1-1362
differential equation solvers
ODE boundary value problems 1-566 1-577
adjusting parameters 1-586
extracting properties 1-582
extracting properties of 1-1512 to 1-1513
1-5649 to 1-5650
forming initial guess 1-583
ODE initial value problems
adjusting parameters of 1-3749
extracting properties of 1-3748
parabolic-elliptic PDE problems 1-3914
`diffuse` 1-1364

DiffuseStrength
  Surface property 1-5129
  surfaceplot property 1-5153
DiffuseStrengthpatch property 1-3863
digamma function 1-4070
dimensions
  size of 1-4758
dir 1-1365
direct term of a partial fraction expansion 1-4503
directories
  copying 1-1094
directory, changing 1-646
discontinuities, eliminating (in arrays of phase angles) 1-5978
discontinuities, plotting functions with 1-1580
discontinuous problems 1-1792
disp
  memmapfile object 1-1375
  serial port I/O 1-1378
  timer object 1-1379
disp, MException method 1-1376
displaying output in Command Window 1-3396
DisplayName
  areaseries property 1-276
  barseries property 1-444
  contourgroup property 1-1052
  errorbarseries property 1-1492
  hggroup property 1-2502
  hgtransform property 1-2531
  image property 1-2619
  Line property 1-3047
  lineseries property 1-3063
  Patch property 1-3863
  quivergroup property 1-4152
  rectangle property 1-4386
  scattergroup property 1-4638
  stairseries property 1-4883
  stemseries property 1-4951
  surface property 1-5130
  surfaceplot property 1-5154
  text property 1-5528
distribution
  Gaussian 1-1475
dither 1-1382
division
  array, left (arithmetic operator) 1-62
  array, right (arithmetic operator) 1-61
  by zero 1-2696
  matrix, left (arithmetic operator) 1-61
  matrix, right (arithmetic operator) 1-61
  of polynomials 1-1288
dlmread 1-1387
dlmwrite 1-1391
dmperm 1-1395
Dockable, Figure property 1-1656
dolly camera 1-601
dot 1-1408
dot product 1-1123 1-1408
dot-parentheses (special characters 1-82
double 1-1409
double click, detecting 1-1683
double integral
  numerical evaluation 1-1224
DoubleBuffer, Figure property 1-1656
dragrect 1-1410
drawing shapes
  circles and rectangles 1-4375
DrawMode, Axes property 1-359
drawnow 1-1412
dsearchn 1-1414
Dulmage-Mendelsohn decomposition 1-1395
dynamic fields 1-82
dynamicprops class 1-1415
dynamicprops.addprop 1-173

E

echo 1-1416
Echo, Root property 1-4541
echoing

functions 1-1416
edge finding, Sobel technique 1-1077
`EdgeAlpha`
patch property 1-3864
surface property 1-5130
surfaceplot property 1-5154
`EdgeColor`
annotation ellipse property 1-232
annotation rectangle property 1-238
annotation textbox property 1-252
areaseries property 1-277
barseries property 1-445
patch property 1-3865
Surface property 1-5131
surfaceplot property 1-5155
Text property 1-5529
`EdgeColor`, rectangle property 1-4387
`EdgeLighting`
patch property 1-3865
Surface property 1-5132
surfaceplot property 1-5156
editable text 1-5726
`Editing`
Text property 1-5530
`eig` 1-1426
eigensystem
transforming 1-653
eigenvalue
accuracy of 1-1426
complex 1-653
matrix logarithm and 1-3142
modern approach to computation of 1-3992
of companion matrix 1-1009
problem 1-1427 1-3997
problem, generalized 1-1427 1-3997
problem, polynomial 1-3997
repeated 1-1428
Wilkinson test matrix and 1-6154
eigenvalues
effect of roundoff error 1-388
improving accuracy 1-388
eigenvector
left 1-1427
matrix, generalized 1-4169
right 1-1427
`eigs` 1-1429
elevation (spherical coordinates) 1-4813
elevation of viewpoint 1-6054
`ellipj` 1-1439
`ellipke` 1-1441
ellipsoid 1-1443
elliptic functions, Jacobian
(defined) 1-1439
elliptic integral
complete (defined) 1-1441
modulus of 1-1439 1-1441
`else` 1-1445
`elseif` 1-1446
`Enable`
Uicontrol property 1-5744
Uimenu property 1-5798
Uipushtool property 1-5844
Uitable property 1-5893
Uitogglehtool property 1-5913
`end` 1-1456
end caps for isosurfaces 1-2879
end of line, indicating 1-83
`eomday` 1-1465
`eq` 1-1472
`eq`, MException method 1-1474
equal arrays
detecting 1-2830 1-2838
equal sign (special characters) 1-81
equations, linear
accuracy of solution 1-1019
`EraseMode`
areaseries property 1-277
barseries property 1-445
contour property 1-1053
errorbar property 1-1493

hggroup property 1-2503
hgtransform property 1-2531
Image property 1-2620
Line property 1-3048
lineseries property 1-3064
quivergroup property 1-4153
rectangle property 1-4387
scatter property 1-4639
stairseries property 1-4883
stem property 1-4952
Surface property 1-5132
surfaceplot property 1-5156
Text property 1-5530
EraseModepatch property 1-3866
error 1-1480
roundoff. *See* roundoff error
error function
complementary 1-1475
(defined) 1-1475
scaled complementary 1-1475
error message
displaying 1-1480
Index into matrix is negative or zero 1-3138
retrieving last generated 1-2957 1-2965
error messages
Out of memory 1-3813
error tolerance
BVP problems 1-587
DDE problems 1-1268
ODE problems 1-3750
errorbars, confidence interval 1-1485
errordlg 1-1507
ErrorMessage, Root property 1-4541
errors
MException class 1-1474
addCause 1-134
constructor 1-3335
disp 1-1376
eq 1-1474
getReport 1-2000
isequal 1-2837
last 1-2954
ne 1-3461
rethrow 1-4510
throw 1-5591
throwAsCaller 1-5595
ErrorType, Root property 1-4542
etree 1-1512
etreeplot 1-1513
evalc 1-1516
evalin 1-1517
event location (DDE) 1-1273
event location (ODE) 1-3757
event.EventData 1-1519
event.listener 1-1520
event.PropertyEvent 1-1522
event.proplistener 1-1523
events 1-1527
examples
calculating isosurface normals 1-2886
contouring mathematical expressions 1-1552
isosurface end caps 1-2879
isosurfaces 1-2890
mesh plot of mathematical function 1-1561
mesh/contour plot 1-1565
plotting filled contours 1-1556
plotting function of two variables 1-1570 1-1573
plotting parametric curves 1-1573
polar plot of function 1-1576
reducing number of patch faces 1-4398
reducing volume data 1-4401
subsampling volume data 1-5097
surface plot of mathematical function 1-1580
surface/contour plot 1-1588
Excel spreadsheets
loading 1-6212
exclamation point (special characters) 1-83
Execute 1-1529
executing statements repeatedly 1-1802

executing statements repeatedly in
  parallel 1-3829
execution
  pausing function 1-3889
  resuming from breakpoint 1-1222
  time for files 1-4056
`exifread` 1-1531
`exist` 1-1532
`exit` 1-1536
`expint` 1-1538
`expm` 1-1539
`expm1` 1-1541
exponential 1-1537
  complex (defined) 1-1537
  integral 1-1538
  matrix 1-1539
exponentiation
  array (arithmetic operator) 1-62
  matrix (arithmetic operator) 1-62
`export2wsdlg` 1-1542
extension, filename
  `.mat` 1-4576
`Extent`
  Text property 1-5532
  Uicontrol property 1-5745
  Uitable property 1-5894
`ezcontour` 1-1551
`ezcontourf` 1-1555
`ezmesh` 1-1559
`ezmeshc` 1-1563
`ezplot` 1-1567
`ezplot3` 1-1572
`ezpolar` 1-1575
`ezsurf` 1-1578
`ezsurfc` 1-1586

F

`FaceAlpha`
  annotation textbox property 1-253
`FaceAlpha`patch property 1-3867
`FaceAlpha`surface property 1-5133
`FaceAlpha`surfaceplot property 1-5158
`FaceColor`
  annotation ellipse property 1-232
  annotation rectangle property 1-238
  areaseries property 1-279
  barseries property 1-446
  Surface property 1-5134
  surfaceplot property 1-5158
`FaceColor`, rectangle property 1-4388
`FaceColor`patch property 1-3868
`FaceLighting`
  Surface property 1-5134
  surfaceplot property 1-5159
`FaceLighting`patch property 1-3868
faces, reducing number in patches 1-4397
`Faces`,patch property 1-3869
`FaceVertexAlphaData`, patch property 1-3870
`FaceVertexCData`,patch property 1-3870
factorization
  LU 1-3166
  QZ 1-3998 1-4169
factorization, Cholesky 1-899
  preordering for 1-1001
`false` 1-1598
`fclose`
  serial port I/O 1-1600
`feather` 1-1601
`feval` 1-1608
`Feval` 1-1610
`fft` 1-1615
FFT. *See* Fourier transform
`fft2` 1-1620
`fftn` 1-1622
`fftshift` 1-1624
`fftw` 1-1627
FFTW 1-1618
`fgetl`
  serial port I/O 1-1633

`fgets`
  serial port I/O 1-1637
field names of a structure, obtaining 1-1640
`fieldnames` 1-1640
fields, of structures
  dynamic 1-82
`figure` 1-1642
Figure
  creating 1-1642
  defining default properties 1-1644
  properties 1-1645
  redrawing 1-4403
figure windows
  moving in front of MATLAB® desktop 1-4744
figure windows, displaying 1-1742
`figurepalette` 1-1703
figures
  annotating 1-3967
  saving 1-4587
Figures
  updating from file 1-1412
file
  extension, getting 1-1720
  modification date 1-1365
file formats
  getting list of supported formats 1-2641
  reading 1-1155 1-2655
  writing 1-2669
file size
  querying 1-2638
`filebrowser` 1-1717
`filemarker` 1-1718
filename
  parts 1-1720
  temporary 1-5220
filename extension
  `.mat` 1-4576
`fileparts` 1-1720
files
  ASCII delimited
    reading 1-1387
    writing 1-1391
  checking existence of 1-1532
  contents, listing 1-5684
  copying 1-1094
  copying with `copyfile` 1-1094
  debugging with `profile` 1-4056
  deleting 1-1330
  Excel spreadsheets
    loading 1-6212
  `fig` 1-4587
  figure, saving 1-4587
  line numbers, listing 1-1245
  listing 1-1365
    in folder 1-6122
  listing contents of 1-5684
  `mdl` 1-4587
  model, saving 1-4587
  opening in Windows applications 1-6155
  optimizing 1-4056
  path, getting 1-1720
  reading
    data from 1-5554
    formatted 1-1847
  reading data from 1-1155
  reading image data from 1-2655
  size, determining 1-1367
  sound
    reading 1-328 1-6102
    writing 1-330 to 1-331 1-6108
  startup 1-3209
  `.wav`
    reading 1-6102
    writing 1-6108
  WK1
    loading 1-6160
    writing to 1-6163
  writing image data to 1-2669
`filesep` 1-1724
`fill` 1-1725

`Fill`
  contour property 1-1054
`fill3` 1-1728
`filter` 1-1731
  digital 1-1731
  finite impulse response (FIR) 1-1731
  infinite impulse response (IIR) 1-1731
  two-dimensional 1-1075
`filter2` 1-1734
`find` 1-1736
`findall` function 1-1741
`findfigs` 1-1742
finding 1-1736
  sign of array elements 1-4752
  See also detecting
`findobj` 1-1743
`findobj` handle method 1-1747
`findprop` handle method 1-1750
`findstr` 1-1752
`finish` 1-1753
`finish.m` 1-4138
FIR filter 1-1731
`FitBoxToText`, annotation textbox
  property 1-253
`FitHeightToText`
  annotation textbox property 1-253
`fitsinfo` 1-1757
`fix` 1-1773
fixed-width font
  axes 1-360
  text 1-5533
  uicontrols 1-5746
  uitables 1-5895
`FixedColors`, Figure property 1-1657
`FixedWidthFontName`, Root property 1-4542
flints 1-3421
flip
  array dimension 1-1777
flip array
  along dimension 1-1777
flip matrix
  on horizontal axis 1-1779
  on vertical axis 1-1778
`flipdim` 1-1777
`fliplr` 1-1778
`flipud` 1-1779
floating-point
  integer, maximum 1-521
floating-point arithmetic, IEEE
  smallest postive number 1-4368
`floor` 1-1781
flow control
  `break` 1-541
  `end` 1-1456
  `error` 1-1482
  `for` 1-1802
  `keyboard` 1-2950
  `parfor` 1-3829
  `return` 1-4514
`fminbnd` 1-1783
`fminsearch` 1-1788
folder
  listing MATLAB files in 1-6122
  root 1-3211
  temporary
    system 1-5219
folders
  checking existence of 1-1532
  copying 1-1094
  creating 1-3355
  listing contents of 1-1365
font
  fixed-width, axes 1-360
  fixed-width, text 1-5533
  fixed-width, uicontrols 1-5746
  fixed-width, uitables 1-5895
`FontAngle`
  annotation textbox property 1-255
  Axes property 1-360
  Text property 1-242 1-5532

Uicontrol property 1-5746
Uitable property 1-5895
`FontName`
annotation textbox property 1-255
Axes property 1-360
Text property 1-5532
textarrow property 1-242
Uicontrol property 1-5746
Uitable property 1-5895
fonts
bold 1-242 1-256 1-5533
specifying size 1-5533
TeX characters
bold 1-5548
italics 1-5548
specifying family 1-5548
specifying size 1-5548
units 1-242 1-256 1-5534
`FontSize`
annotation textbox property 1-256
Axes property 1-361
Text property 1-5533
textarrow property 1-242
Uicontrol property 1-5747
Uitable property 1-5896
`FontUnits`
Axes property 1-361
Text property 1-5534
Uicontrol property 1-5747
Uitable property 1-5896
`FontWeight`
annotation textbox property 1-256
Axes property 1-361
Text property 1-5533
textarrow property 1-242
Uicontrol property 1-5747
Uitable property 1-5896
`fopen`
serial port I/O 1-1800
`for` 1-1802
`ForegroundColor`
Uicontrol property 1-5748
Uimenu property 1-5798
Uitable property 1-5897
`Format` 1-4542
`FormatSpacing`, Root property 1-4543
formatted data
reading from file 1-1847
Fourier transform
algorithm, optimal performance of 1-1618 1-2588 1-2590
as method of interpolation 1-2783
discrete, n-dimensional 1-1622
discrete, one-dimensional 1-1615
discrete, two-dimensional 1-1620
fast 1-1615
inverse, n-dimensional 1-2592
inverse, one-dimensional 1-2588
inverse, two-dimensional 1-2590
shifting the zero-frequency component of 1-1625
`fplot` 1-1813 1-1829
`fprintf`
serial port I/O 1-1826
fraction, continued 1-4201
fragmented memory 1-3813
`frame2im` 1-1829
frames 1-5726
`fread`
serial port I/O 1-1838
`freqspace` 1-1845
frequency response
desired response matrix
frequency spacing 1-1845
frequency vector 1-3145
`fromName` `meta.class` method 1-3286
`fromName` `meta.package` method 1-3309
`fscanf`
serial port I/O 1-1852
`full` 1-1861

func2str 1-1864
function
  echoing commands 1-1416
function handle 1-1869
function handles
  overview of 1-1869
function syntax 1-5206
functions 1-1873
  call history 1-4061
  call stack for 1-1229
  checking existence of 1-1532
  debugging 1-2950
  finding using keywords 1-3146
  help for 1-2464
  in memory 1-2703
  locking (preventing clearing) 1-3376
  pausing execution of 1-3889
  that work down the first non-singleton dimension 1-4745
  unlocking (allowing clearing) 1-3432
funm 1-1877
fwrite
  serial port I/O 1-1886

G

gallery 1-1900
gamma function
  (defined) 1-1927
  incomplete 1-1927
  logarithm of 1-1927
  logarithmic derivative 1-4070
Gauss-Kronrod quadrature 1-4126
Gaussian distribution function 1-1475
Gaussian elimination
  (as algorithm for solving linear equations) 1-2806
  Gauss Jordan elimination with partial pivoting 1-4570
  LU factorization 1-3166
gca 1-1933
gcbf function 1-1934
gcbo function 1-1935
gcf 1-1940
gco 1-1941
ge 1-1942
generalized eigenvalue problem 1-1427 1-3997
genvarname 1-1946
geodesic dome 1-5203
get 1-1950
  memmapfile object 1-1960
  serial port I/O 1-1966
  timer object 1-1968
get (tscollection) 1-1970
get hgsetget class method 1-1959
get, RandStream method 1-1965
getabstime (tscollection) 1-1971
getAllPackages meta.package method 1-3310
getappdata function 1-1973
getCompilerConfigurations 1-3329
getDefaultStream, RandStream method 1-1978 1-4198
getdisp hgsetget class method 1-1979
getenv 1-1980
getframe 1-1986
  image resolution and 1-1987
getpixelposition 1-1993
getpref function 1-1995
getReport, MException method 1-2000
getsampleusingtime (tscollection) 1-2003
gettimeseriesnames 1-2006
gettsafteratevent 1-2007
gettsafterevent 1-2008
gettsatevent 1-2009
gettsbeforeatevent 1-2010
gettsbeforeevent 1-2011
gettsbetweenevents 1-2012
GIF files
  writing 1-2671
ginput function 1-2018

`global` 1-2021
global variable
  defining 1-2021
`gmres` 1-2025
golden section search 1-1786
Goup
  defining default properties 1-2525
`gplot` 1-2039
`grabcode` function 1-2041
`gradient` 1-2043
gradient, numerical 1-2043
graph
  adjacency 1-1396
graph theory 1-5962
graphics objects
  Axes 1-338
  Figure 1-1642
  getting properties 1-1950
  Image 1-2603
  Light 1-3021
  Line 1-3035
  Patch 1-3835
  resetting properties 1-4498
  Root 1-4538
  setting properties 1-4682
  Surface 1-5116
  Text 1-5516
  uicontextmenu 1-5713
  Uicontrol 1-5725
  Uimenu 1-5787
graphics objects, deleting 1-1330
graphs
  editing 1-3967
`graymon` 1-2046
Greek letters and mathematical symbols 1-246
  1-258 1-5546
`grid` 1-2047
grid arrays
  for volumetric plots 1-3278
  multi-dimensional 1-3454
`griddatan` 1-2056
`GridLineStyle`, Axes property 1-362
group
  `hggroup` function 1-2495
`gsvd` 1-2066
`gt` 1-2072
`gtext` 1-2074
`guidata` function 1-2075
GUIDE
  object methods
    `inspect` 1-2723
`guihandles` function 1-2080
GUIs, printing 1-4032
`gunzip` 1-2081
`gzip` 1-2083

H

`hadamard` 1-2428
Hadamard matrix 1-2428
  subspaces of 1-5092
handle class 1-2429
handle graphics
  `hgtransform` 1-2516
handle graphics`hggroup` 1-2495
handle relational operators 1-4485
`handle.addlistener` 1-159
`handle.delete` 1-1334
`handle.findobj` 1-1747
`handle.findprop` 1-1750
`handle.isvalid` 1-2920
`handle.notify` 1-3611
`HandleVisibility`
  areaseries property 1-279
  Axes property 1-362
  barseries property 1-447
  contour property 1-1054
  errorbar property 1-1494
  Figure property 1-1658
  hggroup property 1-2505

hgtransform property 1-2533
Image property 1-2621
Light property 1-3027
Line property 1-3049
lineseries property 1-3065
patch property 1-3872
quivergroup property 1-4154
rectangle property 1-4388
Root property 1-4543
stairseries property 1-4885
stem property 1-4953
Surface property 1-5135
surfaceplot property 1-5159
Text property 1-5534
Uicontextmenu property 1-5720
Uicontrol property 1-5748
Uimenu property 1-5798
Uipushtool property 1-5844
Uitable property 1-5897
Uitoggletool property 1-5913
Uitoolbar property 1-5926

`hankel` 1-2432
Hankel matrix 1-2432
HDF
  appending to when saving (`WriteMode`) 1-2675
  compression 1-2675
  setting JPEG quality when writing 1-2675
HDF files
  writing images 1-2671
HDF4
  summary of capabilities 1-2433
`hdf5info` 1-2436
`hdf5read` 1-2438
`hdf5write` 1-2440
`hdfinfo` 1-2444
`hdfread` 1-2452
`hdftool` 1-2463
`Head1Length`
  annotation doublearrow property 1-227
`Head1Style`
  annotation doublearrow property 1-228
`Head1Width`
  annotation doublearrow property 1-229
`Head2Length`
  annotation doublearrow property 1-227
`Head2Style`
  annotation doublearrow property 1-229
`Head2Width`
  annotation doublearrow property 1-229
`HeadLength`
  annotation arrow property 1-223
  textarrow property 1-243
`HeadStyle`
  annotation arrow property 1-223
  textarrow property 1-243
`HeadWidth`
  annotation arrow property 1-224
  textarrow property 1-244
`Height`
  annotation ellipse property 1-232
`help` 1-2464
  keyword search in functions 1-3146
  online 1-2464
`helpbrowser` 1-2467
`helpdesk` 1-2468
`helpdlg` 1-2469
`helpwin` 1-2471
`hess` 1-2472
Hessenberg form of a matrix 1-2472
`hex2dec` 1-2492
`hex2num` 1-2493
`hgsetget` class 1-2515
`hgsetget.get` 1-1959
`hgsetget.getdisp` 1-1979
`hgsetget.set` 1-4691
`hidden` 1-2539
Hierarchical Data Format (HDF) files
  writing images 1-2671
`hilb` 1-2542

Hilbert matrix 1-2542
  inverse 1-2808
histc 1-2557
`HitTest`
  areaseries property 1-280
  Axes property 1-363
  barseries property 1-448
  contour property 1-1056
  errorbar property 1-1496
  Figure property 1-1659
  hggroup property 1-2506
  hgtransform property 1-2534
  Image property 1-2623
  Light property 1-3028
  Line property 1-3050
  lineseries property 1-3067
  Patch property 1-3873
  quivergroup property 1-4155
  rectangle property 1-4389
  Root property 1-4543
  scatter property 1-4642
  stairseries property 1-4886
  stem property 1-4955
  Surface property 1-5136
  surfaceplot property 1-5161
  Text property 1-5535
  Uicontrol property 1-5749
  Uipushtool property 1-5845
  Uitable property 1-5898
  Uitoggletool property 1-5914
  Uitoolbarl property 1-5927
`HitTestArea`
  areaseries property 1-281
  barseries property 1-448
  contour property 1-1056
  errorbar property 1-1496
  quivergroup property 1-4156
  scatter property 1-4642
  stairseries property 1-4886
  stem property 1-4955
`hold` 1-2560
`home` 1-2562
`HorizontalAlignment`
  Text property 1-5536
  textbox property 1-244 1-256
  Uicontrol property 1-5749
`horzcat` 1-2563
`horzcat` (function equivalent for `[,]`) 1-83
`horzcat (tscollection)` 1-2565
`hostid` 1-2566
`hsv2rgb` 1-2567
hyperbolic
  cosecant 1-1127
  cosecant, inverse 1-107
  cosine 1-1105
  cosine, inverse 1-99
  cotangent 1-1109
  cotangent, inverse 1-103
  secant 1-4668
  secant, inverse 1-298
  sine 1-4757
  sine, inverse 1-303
  tangent 1-5216
  tangent, inverse 1-315
hyperplanes, angle between 1-5092
`hypot` 1-2568

I

`i` 1-2571
icon images
  reading 1-2657
identity matrix
  sparse 1-4810
`idivide` 1-2583
IEEE floating-point arithmetic
  smallest positive number 1-4368
`ifft` 1-2588
`ifft2` 1-2590
`ifftn` 1-2592

`ifftshift` 1-2594
IIR filter 1-1731
`ilu` 1-2595
`im2java` 1-2600
`imag` 1-2602
`image` 1-2603
Image
    creating 1-2603
    properties 1-2611
image types
    querying 1-2639
images
    file formats 1-2655 1-2669
    reading data from files 1-2655
    writing to files 1-2669
Images
    converting MATLAB image to Java
        Image 1-2600
`imagesc` 1-2628
imaginary 1-2602
    part of complex number 1-2602
    unit (`sqrt(\xd0 1)`) 1-2571 1-2925
    See also complex
`imapprox` 1-2633
`imformats` 1-2641
import 1-2644
importing
    Java class and package names 1-2644
`imread` 1-2655
`imwrite` 1-2669
incomplete beta function
    (defined) 1-485
incomplete gamma function
    (defined) 1-1928
`ind2sub` 1-2692
Index into matrix is negative or zero (error
    message) 1-3138
indexed images
    converting from RGB 1-4517
indexing
    logical 1-3137
indices, array
    of sorted elements 1-4781
`Inf` 1-2696
infinity 1-2696
`info` 1-2699
`inline` 1-2700
`inmem` 1-2703
`inpolygon` 1-2705
input
    checking number of arguments 1-3437
    name of array passed as 1-2715
`inputdlg` 1-2710
`inputname` 1-2715
`inspect` 1-2723
installation, root folder 1-3211
instance properties 1-173
`instrcallback` 1-2731
`instrfind` 1-2732
`instrfindall` 1-2734
    example of 1-2735
`int2str` 1-2736
integer
    floating-point, maximum 1-521
`IntegerHandle`
    Figure property 1-1659
integration
    polynomial 1-4003
    quadrature 1-4111 1-4122
`interp1` 1-2768
`interp1q` 1-2774
`interp2` 1-2776
`interp3` 1-2780
`interpft` 1-2783
`interpn` 1-2784
interpolated shading and printing 1-4033
interpolation
    cubic method 1-2769 1-2777 1-2780 1-2785
    cubic spline method 1-2769 1-2777 1-2780
        1-2785

FFT method 1-2783
linear method 1-2769 1-2777 1-2780 1-2785
multidimensional 1-2784
nearest neighbor method 1-2769 1-2777 1-2780 1-2785
one-dimensional 1-2768
three-dimensional 1-2780
two-dimensional 1-2776

`Interpreter`
Text property 1-5537
textarrow property 1-244
textbox property 1-256

interpstreamspeed 1-2788

`Interruptible`
areaseries property 1-281
Axes property 1-363
barseries property 1-449
contour property 1-1056
errorbar property 1-1496
Figure property 1-1659
hggroup property 1-2506
hgtransform property 1-2534
Image property 1-2623
Light property 1-3028
Line property 1-3050
lineseries property 1-3067
patch property 1-3873
quivergroup property 1-4156
rectangle property 1-4390
Root property 1-4543
scatter property 1-4643
stairseries property 1-4887
stem property 1-4955
Surface property 1-5136 1-5161
Text property 1-5538
Uicontextmenu property 1-5721
Uicontrol property 1-5750
Uimenu property 1-5799
Uipushtool property 1-5845
Uitable property 1-5898
Uitoggletool property 1-5914
Uitoolbar property 1-5927

`intersect` 1-2792
`intmax` 1-2804
`intmin` 1-2805
`inv` 1-2806

inverse
cosecant 1-104
cosine 1-96
cotangent 1-100
Fourier transform 1-2588 1-2590 1-2592
Hilbert matrix 1-2808
hyperbolic cosecant 1-107
hyperbolic cosine 1-99
hyperbolic cotangent 1-103
hyperbolic secant 1-298
hyperbolic sine 1-303
hyperbolic tangent 1-315
of a matrix 1-2806
secant 1-295
tangent 1-308
tangent, four-quadrant 1-309

inversion, matrix
accuracy of 1-1019

`InvertHardCopy`, Figure property 1-1660
`invhilb` 1-2808
involutary matrix 1-3834
`ipermute` 1-2812
`is*` 1-2813
`isa` 1-2816
`isappdata` function 1-2819
`iscell` 1-2820
`iscellstr` 1-2821
`ischar` 1-2822
`isdir` 1-2825
`isempty` 1-2828
`isempty (tscollection)` 1-2829
`isequal` 1-2830
`isequal`, MException method 1-2837
`isequalwithequalnans` 1-2838

isfield 1-2842
isfinite 1-2844
isfloat 1-2845
isglobal 1-2846
ishandle 1-2848
ishghandle 1-2849
isinf 1-2851
isinteger 1-2852
iskeyword 1-2858
isletter 1-2860
islogical 1-2861
ismac 1-2862
ismember 1-2864
isnan 1-2874
isnumeric 1-2875
isocap 1-2879
isonormals 1-2886
isosurface 1-2889
    calculate data from volume 1-2889
    end caps 1-2879
    vertex normals 1-2886
ispc 1-2894
ispref function 1-2895
isreal 1-2901
isscalar 1-2905
issorted 1-2906
isspace 1-2909 1-2912
issparse 1-2910
isstr 1-2911
isstruct 1-2916
isstudent 1-2917
isunix 1-2919
isvalid 1-2921
    timer object 1-2922
isvalid handle method 1-2920
isvarname 1-2923
isvector 1-2924
italics font
    TeX characters 1-5548

J

j 1-2925
Jacobi rotations 1-4831
Jacobian elliptic functions
    (defined) 1-1439
Jacobian matrix (BVP) 1-589
Jacobian matrix (ODE) 1-3758
    generating sparse numerically 1-3759
        1-3761
    specifying 1-3759 1-3761
    vectorizing ODE function 1-3759 to 1-3761
Java
    class names 1-940 1-2644
    object methods
        inspect 1-2723
Java Image class
    creating instance of 1-2600
Java import list
    adding to 1-2644
    clearing 1-940
Java version used by MATLAB 1-6032
joining arrays. *See* concatenation
Joint Photographic Experts Group (JPEG)
    writing 1-2671
JPEG
    setting Bitdepth 1-2675
    specifying mode 1-2676
JPEG 2000
    setting tile size 1-2677
JPEG 2000 comment
    setting when writing a JPEG 2000
        image 1-2676
    specifying 1-2676
JPEG comment
    setting when writing a JPEG image 1-2675
JPEG files
    parameters that can be set when
        writing 1-2675
    writing 1-2671
JPEG quality

setting when writing a JPEG image 1-2676 to 1-2677 1-2681
setting when writing an HDF image 1-2675
JPEG2000 files
parameters that can be set when writing 1-2676
jvm
version used by MATLAB 1-6032

K

K>> prompt
keyboard function 1-2950
keep
some variables when clearing 1-943
keyboard 1-2950
keyboard mode 1-2950
terminating 1-4514
KeyPressFcn
Uicontrol property 1-5752
Uitable property 1-5900
KeyPressFcn, Figure property 1-1661
KeyReleaseFcn, Figure property 1-1664
keyword search in functions 1-3146
keywords
iskeyword function 1-2858
kron 1-2952
Kronecker tensor product 1-2952
Krylov subspaces 1-5585

L

Label, Uimenu property 1-5800
labeling
plots (with numeric values) 1-3625
LabelSpacing
contour property 1-1058
Laplacian 1-1289
Laplacian matrix 1-5962
largest array elements 1-3232
last, MException method 1-2954
lasterr 1-2957
lasterror 1-2960
lastwarn 1-2965
LaTeX, see TeX 1-246 1-258 1-5546
Layer, Axes property 1-365
Layout Editor
starting 1-2079
LData
errorbar property 1-1498
LDataSource
errorbar property 1-1498
ldivide (function equivalent for .\) 1-65
le 1-2977
least squares
polynomial curve fitting 1-3999
problem, overdetermined 1-3944
legend 1-2979
properties 1-2984
setting text properties 1-2984
legendre 1-2992
Legendre functions
(defined) 1-2992
Schmidt semi-normalized 1-2992
length
serial port I/O 1-2999
length (tscollection) 1-3000
LevelList
contour property 1-1058
LevelListMode
contour property 1-1058
LevelStep
contour property 1-1059
LevelStepMode
contour property 1-1059
license 1-3017
light 1-3021
Light
creating 1-3021
defining default properties 1-2610 1-3022

properties 1-3023
Light object
positioning in spherical coordinates 1-3032
`lightangle` 1-3032
`lighting` 1-3033
limits of axes, setting and querying 1-6198 1-6202 1-6206
`line` 1-3035
editing 1-3967
Line
creating 1-3035
defining default properties 1-3039
properties 1-3041 1-3059
line numbers in files 1-1245
linear audio signal 1-3034 1-3421
linear dependence (of data) 1-5092
linear equation systems
accuracy of solution 1-1019
linear equation systems, methods for solving
matrix inversion (inaccuracy of) 1-2806
linear interpolation 1-2769 1-2777 1-2780 1-2785
linear regression 1-3999
linearly spaced vectors, creating 1-3108
`LineColor`
contour property 1-1059
lines
computing 2-D stream 1-4988
computing 3-D stream 1-4990
drawing stream lines 1-4992
LineSpec 1-3077
`LineStyle`
annotation arrow property 1-225
annotation doublearrow property 1-229
annotation ellipse property 1-232
annotation line property 1-235
annotation rectangle property 1-239
annotation textbox property 1-257
areaseries property 1-282
barseries property 1-450
contour property 1-1060
errorbar property 1-1498
Line property 1-3052
lineseries property 1-3068
patch property 1-3874
quivergroup property 1-4157
rectangle property 1-4391
stairseries property 1-4888
stem property 1-4957
surface object 1-5138
surfaceplot object 1-5162
text object 1-5540
textarrow property 1-245
`LineStyleOrder`
Axes property 1-365
`LineWidth`
annotation arrow property 1-225
annotation doublearrow property 1-230
annotation ellipse property 1-233
annotation line property 1-236
annotation rectangle property 1-239
annotation textbox property 1-257
areaseries property 1-283
Axes property 1-366
barseries property 1-451
contour property 1-1060
errorbar property 1-1499
Line property 1-3052
lineseries property 1-3069
Patch property 1-3875
quivergroup property 1-4158
rectangle property 1-4391
scatter property 1-4644
stairseries property 1-4889
stem property 1-4957
Surface property 1-5138
surfaceplot property 1-5163
text object 1-5541
textarrow property 1-245
`linkaxes` 1-3082
`linkdata` 1-3091

linkprop 1-3099
linsolve 1-3105
linspace 1-3108
list boxes 1-5726
  defining items 1-5759
list, RandStream method 1-3109
ListboxTop, Uicontrol property 1-5753
listdlg 1-3111
listfonts 1-3114
load 1-3116 1-3121
  serial port I/O 1-3123
Lobatto IIIa ODE solver 1-575 1-581
local variables 1-2021
locking functions 1-3376
log 1-3133
  saving session to file 1-1360
log10 [log010] 1-3134
log1p 1-3135
log2 1-3136
logarithm
  base ten 1-3134
  base two 1-3136
  complex 1-3133 to 1-3134
  natural 1-3133
  of beta function (natural) 1-488
  of gamma function (natural) 1-1932
  of real numbers 1-4366
  plotting 1-3139
logarithmic derivative
  gamma function 1-4070
logarithmically spaced vectors, creating 1-3145
logical 1-3137
logical array
  converting numeric array to 1-3137
  detecting 1-2861
logical indexing 1-3137
logical operations
  XOR 1-6233
logical operators 1-71 1-78
logical tests
  all 1-200
  any 1-263
logical XOR 1-6233
loglog 1-3139
logm 1-3142
logspace 1-3145
lookfor 1-3146
lossy compression
  writing JPEG 2000 files with 1-2676
  writing JPEG files with 1-2676
Lotus WK1 files
  loading 1-6160
  writing 1-6163
lower 1-3148
lower triangular matrix 1-5652
lowercase to uppercase 1-5983
lscov 1-3151
lsqnonneg 1-3156
lsqr 1-3159
lt 1-3164
lu 1-3166
LU factorization 1-3166
  storage requirements of (sparse) 1-3636
luinc 1-3173

M

.m files
  checking existence of 1-1532
M-files
  deleting 1-1330
magic 1-3180
magic squares 1-3180
Margin
  annotation textbox property 1-257
  text object 1-5543
Marker
  errorbar property 1-1499
  Line property 1-3053
  lineseries property 1-3069

Patch property 1-3875
quivergroup property 1-4158
scatter property 1-4644
stairseries property 1-4889
stem property 1-4958
Surface property 1-5138
surfaceplot property 1-5163
`MarkerEdgeColor`
errorbar property 1-1500
Line property 1-3053
lineseries property 1-3070
Patch property 1-3876
quivergroup property 1-4159
scatter property 1-4645
stairseries property 1-4890
stem property 1-4958
Surface property 1-5139
surfaceplot property 1-5164
`MarkerFaceColor`
errorbar property 1-1500
Line property 1-3054
lineseries property 1-3070
Patch property 1-3876
quivergroup property 1-4159
scatter property 1-4645
stairseries property 1-4890
stem property 1-4959
Surface property 1-5140
surfaceplot property 1-5164
`MarkerSize`
errorbar property 1-1501
Line property 1-3054
lineseries property 1-3071
Patch property 1-3877
quivergroup property 1-4160
stairseries property 1-4891
stem property 1-4959
Surface property 1-5140
surfaceplot property 1-5165
mass matrix (ODE) 1-3762
initial slope 1-3763 to 1-3764
singular 1-3763
sparsity pattern 1-3763
specifying 1-3763
state dependence 1-3763
MAT-file 1-4576
converting sparse matrix after loading from 1-4797
MAT-files
listing for folder 1-6122
`mat2str` 1-3196
`material` 1-3198
MATLAB
installation folder 1-3211
quitting 1-4138
startup 1-3209
version number, comparing 1-6030
version number, displaying 1-6022
`matlab` (UNIX command) 1-3213
`matlab` (Windows command) 1-3226
MATLAB files
listing names of in a folder 1-6122
`matlab` function for UNIX 1-3213
`matlab` function for Windows 1-3226
MATLAB startup file 1-4899
MATLAB® desktop
moving figure windows in front of 1-4744
`matlab.mat` 1-4576
`matlabrc` 1-3209
matlabroot 1-3211
`$matlabroot` 1-3211
matrix 1-60
addressing selected rows and columns of 1-85
arrowhead 1-1001
columns
rearrange 1-1778
companion 1-1009
condition number of 1-1019 1-4206
condition number, improving 1-388

converting to vector 1-86
defective (defined) 1-1428
detecting sparse 1-2910
determinant of 1-1351
diagonal of 1-1356
Dulmage-Mendelsohn decomposition 1-1395
evaluating functions of 1-1877
exponential 1-1539
Hadamard 1-2428 1-5092
Hankel 1-2432
Hermitian Toeplitz 1-5642
Hessenberg form of 1-2472
Hilbert 1-2542
inverse 1-2806
inverse Hilbert 1-2808
inversion, accuracy of 1-1019
involutary 1-3834
left division (arithmetic operator) 1-61
lower triangular 1-5652
magic squares 1-3180 1-5101
modal 1-1426
multiplication (defined) 1-61
Pascal 1-3834 1-4006
permutation 1-3166
poorly conditioned 1-2542
power (arithmetic operator) 1-62
pseudoinverse 1-3944
reading files into 1-1387
rearrange
  columns 1-1778
  rows 1-1779
reduced row echelon form of 1-4570
replicating 1-4493
right division (arithmetic operator) 1-61
rotating 90\xfb 1-4558
rows
  rearrange 1-1779
Schur form of 1-4572 1-4652
singularity, test for 1-1351
sorting rows of 1-4784
sparse. *See* sparse matrix
specialized 1-1900
square root of 1-4845
subspaces of 1-5092
test 1-1900
Toeplitz 1-5642
trace of 1-1356 1-5645
transpose (arithmetic operator) 1-62
transposing 1-82
unitary 1-5182
upper triangular 1-5673
Vandermonde 1-4001
Wilkinson 1-4803 1-6154
writing formatted data to 1-1847
writing to ASCII delimited file 1-1391
writing to spreadsheet 1-6163
See also array
`Matrix`
  hgtransform property 1-2536
matrix functions
  evaluating 1-1877
matrix power. *See* matrix, exponential
`max` 1-3232
`Max`, Uicontrol property 1-5753
`MaxHeadSize`
  quivergroup property 1-4160
maximum matching 1-1395
MDL-files
  checking existence of 1-1532
`mean` 1-3237
`memmapfile` 1-3244
`memory` 1-3250
  minimizing use of 1-3813
  variables in 1-6139
menu (of user input choices) 1-3260
`menu` function 1-3260
`MenuBar`, Figure property 1-1667
Mersenne twister 1-4188 1-4195
mesh plot
  tetrahedron 1-5221

mesh size (BVP) 1-591
meshc 1-3262 1-3268 1-3273
meshgrid 1-3278
MeshStyle, Surface property 1-5140
MeshStyle, surfaceplot property 1-5165
meshz 1-3262 1-3268 1-3273
message
  error See error message 1-6093
  warning See warning message 1-6093
meta.class 1-3281
meta.DynamicProperty 1-3287
meta.EnumeratedValue class 1-3292
meta.event 1-3294
meta.method 1-3300
meta.package class 1-3303
meta.property 1-3311
methods
  for timeseries object 1-4235
mex 1-3321
mex build script
  switches 1-3322
    -arch 1-3322
    -c 1-3322
    -compatibleArrayDims 1-3323
    -cxx 1-3323
    -Dname 1-3323
    -Dname=value 1-3323
    -f optionsfile 1-3323
    -fortran 1-3324
    -g 1-3324
    -h[elp] 1-3324
    -inline 1-3324
    -Ipathname 1-3324
    -largeArrayDims 1-3325
    -Lfolder 1-3324
    -lname 1-3324
    -n 1-3325
    name=value 1-3326
    -O 1-3325
    -outdir dirname 1-3325
    -output resultname 1-3325
    @rsp_file 1-3322
    -setup 1-3325
    -Uname 1-3326
    -v 1-3326
mex.CompilerConfiguration 1-3329
mex.CompilerConfigurationDetails 1-3329
MEX-files
  debugging on UNIX 1-1226
  listing for folder 1-6122
mex.getCompilerConfigurations 1-3329
MException
  constructor 1-1474 1-3335
  methods
    addCause 1-134
    disp 1-1376
    eq 1-1474
    getReport 1-2000
    isequal 1-2837
    last 1-2954
    ne 1-3461
    rethrow 1-4510
    throw 1-5591
    throwAsCaller 1-5595
mexext 1-3341
mfilename 1-3342
Microsoft Excel files
  loading 1-6212
min 1-3344
Min, Uicontrol property 1-5754
MinColormap, Figure property 1-1667
MinorGridLineStyle, Axes property 1-366
minres 1-3348
minus (function equivalent for -) 1-65
mislocked 1-3354
mkdir 1-3355
mkpp 1-3358
mldivide (function equivalent for \) 1-65
mlintrpt 1-3374
  suppressing messages 1-3375

mlock 1-3376
mmfileinfo 1-3377
mod 1-3385
modal matrix 1-1426
mode objects
 pan, using 1-3818
 rotate3d, using 1-4563
 zoom, using 1-6247
models
 saving 1-4587
modification date
 of a file 1-1365
modulo arithmetic 1-3385
MonitorPositions
 Root property 1-4544
Moore-Penrose pseudoinverse 1-3944
more 1-3396 1-3421
move 1-3398
movefile 1-3400
movegui function 1-3403
movie 1-3406
movie2avi 1-3410
movies
 exporting in AVI format 1-331
mpower (function equivalent for ^) 1-65
mrdivide (function equivalent for /) 1-65
msgbox 1-3415
mtimes 1-3418
mtimes (function equivalent for *) 1-65
mu-law encoded audio signals 1-3034 1-3421
multibandread 1-3422
multibandwrite 1-3427
multidimensional arrays
 concatenating 1-636
 interpolation of 1-2784
 number of dimensions of 1-3456
 rearranging dimensions of 1-2812 1-3935
 removing singleton dimensions of 1-4848
 reshaping 1-4501
 size of 1-4758
 sorting elements of 1-4780
multiplication
 array (arithmetic operator) 1-61
 matrix (defined) 1-61
 of polynomials 1-1073
multistep ODE solver 1-3656 1-3671 1-3686
 1-3701 1-3716 1-3731 1-3746
munlock 1-3432

N

Name, Figure property 1-1668
namelengthmax 1-3434
NaN 1-3435
NaN (Not-a-Number) 1-3435
 returned by rem 1-4487
nargchk 1-3437
narginchk 1-3441
nargoutchk 1-3446
native2unicode 1-3448
ndgrid 1-3454
ndims 1-3456
ne 1-3457
ne, MException method 1-3461
nearest neighbor interpolation 1-2769 1-2777
 1-2780 1-2785
NET
 summary of functions 1-3464
.NET
 summary of functions 1-3464
netcdf
 summary of capabilities 1-3576
netcdf.abort
 revert recent netCDF file definitions 1-3517
netcdf.close
 close netCDF file 1-3519
netcdf.copyAtt
 copy attribute to new location 1-3520
netcdf.create
 create netCDF file 1-3522

netcdf.defDim
create dimension in netCDF file 1-3524
netcdf.defVar
define variable in netCDF dataset 1-3526
netcdf.delAtt
delete netCDF attribute 1-3535
netcdf.endDef
takes a netCDF file out of define mode 1-3537
netcdf.getAtt
return data from netCDF attribute 1-3539
netcdf.getConstant
get numeric value of netCDF constant 1-3542
netcdf.getConstantNames
get list of netCDF constants 1-3543
netcdf.getVar
return data from netCDF variable 1-3544
netcdf.inq
return information about netCDF file 1-3547
netcdf.inqAtt
return information about a netCDF attribute 1-3568
netcdf.inqAttID
return identifier of netCDF attribute 1-3570
netcdf.inqAttName
return name of netCDF attribute 1-3571
netcdf.inqDim
return information about netCDF dimension 1-3573
netcdf.inqDimID
return dimension ID for netCDF file 1-3574
netcdf.inqLibVers
return version of netCDF library 1-3575
netcdf.inqVarID
return netCDF variable identifier 1-3578
netcdf.open
open an existing netCDF file 1-3579
netcdf.putAtt
write a netCDF attribute 1-3581
netcdf.putVar
write data to netCDF variable 1-3583
netcdf.reDef
put netCDF file into define mode 1-3586
netcdf.renameAtt
netCDF function to change the name of an attribute 1-3587
netcdf.renameDim
netCDF function to change the name of a dimension 1-3589
netcdf.renameVar
change the name of a netCDF variable 1-3590
netcdf.setDefaultFormat
change the default netCDF file format 1-3594
netcdf.setFill
set netCDF fill behavior 1-3595
netcdf.sync
synchronize netCDF dataset to disk 1-3596
`newplot` 1-3597
`NextPlot`
Axes property 1-366
Figure property 1-1668
`nnz` 1-3604
no derivative method 1-1792
`nodesktop` startup option 1-3218
nonzero entries
specifying maximum number of in sparse matrix 1-4794
nonzero entries (in sparse matrix)
allocated storage for 1-3636
number of 1-3604
replacing with ones 1-4825
vector of 1-3606
`nonzeros` 1-3606
`norm` 1-3607
1-norm 1-4206
2-norm (estimate of) 1-3608
matrix 1-3607
pseudoinverse and 1-3944 1-3946
vector 1-3607
normal vectors, computing for volumes 1-2886

NormalMode
Patch property 1-3877
Surface property 1-5141
surfaceplot property 1-5165
normest 1-3608
not 1-3609
not (function equivalent for ~) 1-75
notebook 1-3610
notify 1-3611
now 1-3612
nthroot 1-3613
null 1-3614
null space 1-3614
num2hex 1-3624
number
of array dimensions 1-3456
numbers
imaginary 1-2602
NaN 1-3435
plus infinity 1-2696
real 1-4365
smallest positive 1-4368
NumberTitle, Figure property 1-1669
numerical differentiation formula ODE
solvers 1-3656 1-3671 1-3686 1-3701 1-3716
1-3731 1-3746
numerical evaluation
double integral 1-1224
triple integral 1-5655
nzmax 1-3636

O

ODE solver properties
error tolerance 1-3750
event location 1-3757
Jacobian matrix 1-3758
mass matrix 1-3762
ode15s 1-3764
solver output 1-3752
step size 1-3755
ODE solvers
backward differentiation formulas 1-3764
numerical differentiation formulas 1-3764
obtaining solutions at specific times 1-3643
1-3658 1-3673 1-3688 1-3703 1-3718
1-3733
variable order solver 1-3764
ode15i function 1-3637
odeget 1-3748
odephas2 output function 1-3754
odephas3 output function 1-3754
odeplot output function 1-3754
odeprint output function 1-3754
odeset 1-3749
odextend 1-3766
off-screen figures, displaying 1-1742
OffCallback
Uitoggletool property 1-5916
OnCallback
Uitoggletool property 1-5917
one-step ODE solver 1-3655
online help 1-2464
openfig 1-3781
OpenGL 1-1676
autoselection criteria 1-1680
opening
files in Windows applications 1-6155
operating system command, issuing 1-83
operators
arithmetic 1-60
logical 1-71 1-78
overloading arithmetic 1-66
overloading relational 1-69
relational 1-69 1-3137
symbols 1-2464
optimget 1-3791
optimization parameters structure 1-3791 to
1-3792
optimizing file execution 1-4056

optimset 1-3792
or 1-3796
or (function equivalent for |) 1-75
ordeig 1-3798
orderfields 1-3801
ordering
  reverse Cuthill-McKee 1-5193 1-5203
ordqz 1-3804
ordschur 1-3806
orient 1-3808
orth 1-3810
orthographic projection, setting and querying 1-615
Out of memory (error message) 1-3813
OuterPosition
  Axes property 1-367
  Figure property 1-1669
output
  checking number of arguments 1-3446
  in Command Window 1-3396
output points (ODE)
  increasing number of 1-3752
output properties (DDE) 1-1269
output properties (ODE) 1-3752
  increasing number of output points 1-3752
overflow 1-2696
overloading
  arithmetic operators 1-66
  relational operators 1-69
  special characters 1-84

P

P-files
  checking existence of 1-1532
pack 1-3813
padecoef 1-3815
pagesetupdlg 1-3816
paging in the Command Window 1-3396
pan mode objects 1-3818
PaperOrientation, Figure property 1-1670
PaperPosition, Figure property 1-1670
PaperPositionMode, Figure property 1-1671
PaperSize, Figure property 1-1671
PaperType, Figure property 1-1671
PaperUnits, Figure property 1-1673
parametric curve, plotting 1-1572
Parent
  areaseries property 1-283
  Axes property 1-368
  barseries property 1-451
  contour property 1-1060
  errorbar property 1-1501
  Figure property 1-1673
  hggroup property 1-2507
  hgtransform property 1-2536
  Image property 1-2624
  Light property 1-3029
  Line property 1-3054
  lineseries property 1-3071
  Patch property 1-3877
  quivergroup property 1-4160
  rectangle property 1-4392
  Root property 1-4547
  scatter property 1-4646
  stairseries property 1-4891
  stem property 1-4959
  Surface property 1-5141
  surfaceplot property 1-5166
  Text property 1-5545
  Uicontextmenu property 1-5723
  Uicontrol property 1-5755
  Uimenu property 1-5801
  Uipushtool property 1-5847
  Uitable property 1-5901
  Uitoggletool property 1-5917
  Uitoolbar property 1-5929
parentheses (special characters) 1-81
parfor 1-3829
partial fraction expansion 1-4503

`pascal` 1-3834
Pascal matrix 1-3834 1-4006
`patch` 1-3835
Patch
    converting a surface to 1-5114
    creating 1-3835
    properties 1-3855
    reducing number of faces 1-4397
    reducing size of face 1-4748
`path` 1-3882
`path2rc` 1-3885
`pathsep` 1-3886
`pathtool` 1-3887
`pause` 1-3889
pauses, removing 1-1219
pausing function execution 1-3889
`pbaspect` 1-3891
PBM
    parameters that can be set when
        writing 1-2677
PBM files
    writing 1-2672
`pcg` 1-3897
`pchip` 1-3902
`pcolor` 1-3909
PCX files
    writing 1-2672
PDE. *See* Partial Differential Equations
`pdepe` 1-3914
`pdeval` 1-3927
percent sign (special characters) 1-83
percent-brace (special characters) 1-83
perfect matching 1-1395
performance 1-462
period (.), to distinguish matrix and array
    operations 1-60
period (special characters) 1-82
`perl` 1-3931
`perl` function 1-3931
Perl scripts in MATLAB 1-3931
permutation
    matrix 1-3166
    of array dimensions 1-3935
    random 1-4185
`permute` 1-3935
`persistent` 1-3936
persistent variable 1-3936
perspective projection, setting and
    querying 1-615
PGM
    parameters that can be set when
        writing 1-2677
PGM files
    writing 1-2672
phase angle, complex 1-216
phase, complex
    correcting angles 1-5975
`pie` 1-3940
`pie3` 1-3942
`pinv` 1-3944
`planerot` 1-3947
`plot`
    editing 1-3967
plot box aspect ratio of axes 1-3891
plot editing mode
    overview 1-3968
Plot Editor
    interface 1-3968
plot, volumetric
    generating grid arrays for 1-3278
    slice plot 1-4769
`PlotBoxAspectRatio`, Axes property 1-369
`PlotBoxAspectRatioMode`, Axes property 1-369
`plotedit` 1-3967
plotting
    3-D plot 1-3963
    contours (a 1-1551
    contours (ez function) 1-1551
    ez-function mesh plot 1-1559
    feather plots 1-1601

filled contours 1-1555
function plots 1-1813
functions with discontinuities 1-1580
in polar coordinates 1-1575
isosurfaces 1-2889
loglog plot 1-3139
mathematical function 1-1567
mesh contour plot 1-1563
mesh plot 1-3262 1-3268 1-3273
parametric curve 1-1572
plot with two y-axes 1-3981
ribbon plot 1-4523
rose plot 1-4554
semilogarithmic plot 1-4670 1-4673
surface plot 1-5106 1-5110
surfaces 1-1578
velocity vectors 1-1023
volumetric slice plot 1-4769
. *See* visualizing
plus (function equivalent for +) 1-65
PNG
writing options for 1-2678
alpha 1-2678
background color 1-2678
chromaticities 1-2679
gamma 1-2679
interlace type 1-2679
resolution 1-2680
significant bits 1-2679
transparency 1-2680
PNG files
writing 1-2672
PNM files
writing 1-2672
Pointer, Figure property 1-1673
PointerLocation, Root property 1-4547
PointerShapeCData, Figure property 1-1674
PointerShapeHotSpot, Figure property 1-1674
PointerWindow, Root property 1-4548
pol2cart 1-3987
polar 1-3989
polar coordinates 1-3987
computing the angle 1-216
converting from Cartesian 1-628
converting to cylindrical or Cartesian 1-3987
plotting in 1-1575
poles of transfer function 1-4503
poly 1-3991
polyarea 1-3994
polyder 1-3996
polyeig 1-3997
polyfit 1-3999
polygamma function 1-4070
polygon
area of 1-3994
creating with patch 1-3835
detecting points inside 1-2705
polyint 1-4003
polynomial
analytic integration 1-4003
characteristic 1-3991 to 1-3992 1-4552
coefficients (transfer function) 1-4503
curve fitting with 1-3999
derivative of 1-3996
division 1-1288
eigenvalue problem 1-3997
evaluation 1-4004
evaluation (matrix sense) 1-4006
make piecewise 1-3358
multiplication 1-1073
polyval 1-4004
polyvalm 1-4006
poorly conditioned
matrix 1-2542
poorly conditioned eigenvalues 1-388
pop-up menus 1-5727
defining choices 1-5759
Portable Anymap files
writing 1-2672
Portable Bitmap (PBM) files

writing 1-2672
Portable Graymap files
writing 1-2672
Portable Network Graphics files
writing 1-2672
Portable pixmap format
writing 1-2672
`Position`
annotation ellipse property 1-233
annotation line property 1-236
annotation rectangle property 1-239
arrow property 1-225
Axes property 1-369
doubletarrow property 1-230
Figure property 1-1674
Light property 1-3029
Text property 1-5545
textarrow property 1-245
textbox property 1-257
Uicontextmenu property 1-5723
Uicontrol property 1-5755
Uimenu property 1-5801
Uitable property 1-5901
position of camera
dollying 1-601
position of camera, setting and querying 1-613
`Position`, rectangle property 1-4392
PostScript
default printer 1-4022
levels 1 and 2 1-4022
printing interpolated shading 1-4033
`pow2` 1-4008
`power` 1-4009
matrix. *See* matrix exponential
of real numbers 1-4369
`power` (function equivalent for `.^`) 1-65
PPM
parameters that can be set when writing 1-2677
PPM files
writing 1-2672
`ppval` 1-4010
`prefdir` 1-4012
`preferences` 1-4014
opening the dialog box 1-4014
present working directory 1-4092
prime factors
dependence of Fourier transform on 1-1618 1-1620 1-1622
`printdlg` 1-4038
`printdlg` function 1-4038
printer
default for linux and unix 1-4022
printer drivers
GhostScript drivers 1-4018
interploated shading 1-4033
MATLAB printer drivers 1-4018
printing
GUIs 1-4032
interpolated shading 1-4033
on MS-Windows 1-4031
with a variable file name 1-4034
with `nodisplay` 1-4025
with noFigureWindows 1-4025
printing figures
preview 1-4039
printing tips 1-4031
printing, suppressing 1-82
`printpreview` 1-4039
product
Kronecker tensor 1-2952
of vectors (cross) 1-1123
scalar (dot) 1-1123
`profile` 1-4056
`profsave` 1-4063
program execution
resuming after suspending 1-5859
suspending from GUI 1-5931
projection type, setting and querying 1-615
`ProjectionType`, Axes property 1-370

prompting users to choose an item 1-3260
propedit 1-4064 to 1-4065
proppanel 1-4068
pseudoinverse 1-3944
psi 1-4070
push buttons 1-5727
pwd 1-4092

Q

qmr 1-4093
QR decomposition
  deleting column from 1-4104
qrdelete 1-4104
qrinsert 1-4106
qrupdate 1-4108
quad 1-4111
quadgk 1-4122
quadl 1-4128
quadrature 1-4111 1-4122
quadv 1-4131
questdlg 1-4134
questdlg function 1-4134
quit 1-4138
quitting MATLAB 1-4138
quiver 1-4141
quiver3 1-4144
qz 1-4169
QZ factorization 1-3998 1-4169

R

radio buttons 1-5727
rand, RandStream method 1-4174
randi, RandStream method 1-4179
randn, RandStream method 1-4184
random
  permutation 1-4185
  sparse matrix 1-4829 to 1-4830
  symmetric sparse matrix 1-4831
random number generators 1-3109 1-4174
  1-4179 1-4184 1-4188 1-4195
randperm 1-4185
randStream
  constructor 1-4195
RandStream 1-4188 1-4195
  constructor 1-4188
  methods
    create 1-1114
    get 1-1965
    getDefaultStream 1-1978 1-4198
    list 1-3109
    rand 1-4174
    randi 1-4179
    randn 1-4184
    setDefaultStream 1-4703
range space 1-3810
rank 1-4200
rank of a matrix 1-4200
RAS files
  parameters that can be set when
    writing 1-2681
  writing 1-2673
RAS image format
  specifying color order 1-2681
  writing alpha data 1-2681
Raster image files
  writing 1-2673
rational fraction approximation 1-4201
rbbox 1-4204 1-4403
rcond 1-4206
rdivide (function equivalent for ./) 1-65
readasync 1-4215
reading
  data from files 1-5554
  formatted data from file 1-1847
readme files, displaying 1-2825
real 1-4365
real numbers 1-4365
reallog 1-4366

realmax 1-4367
realmin 1-4368
realpow 1-4369
realsqrt 1-4370
rearrange array
  flip along dimension 1-1777
  reverse along dimension 1-1777
rearrange matrix
  flip left-right 1-1778
  flip up-down 1-1779
  reverse column order 1-1778
  reverse row order 1-1779
RearrangeableColumns
  Uitable property 1-5902
rearranging arrays
  converting to vector 1-86
  removing first n singleton dimensions 1-4745
  removing singleton dimensions 1-4848
  reshaping 1-4501
  shifting dimensions 1-4745
  swapping dimensions 1-2812 1-3935
rearranging matrices
  converting to vector 1-86
  rotating 90\xfb 1-4558
  transposing 1-82
record 1-4371
rectangle
  properties 1-4380
  rectangle function 1-4375
rectint 1-4394
RecursionLimit
  Root property 1-4548
recycle 1-4395
reduced row echelon form 1-4570
reducepatch 1-4397
reducevolume 1-4401
refresh 1-4403
regexptranslate 1-4477
regression
  linear 1-3999
regular expression operators
  conditional operators
    if condition, match expr
      ((?(condition)expr)) 1-4413 1-4442
      1-4465
regularly spaced vectors, creating 1-85 1-3108
rehash 1-4482
relational operators 1-69 1-3137
relational operators for handle objects 1-4485
relative accuracy
  BVP 1-587
  DDE 1-1269
  norm of DDE solution 1-1269
  norm of ODE solution 1-3751
  ODE 1-3751
rem 1-4487
removets 1-4490
renaming
  using copyfile 1-1094
renderer
  OpenGL 1-1676
  painters 1-1676
  zbuffer 1-1676
Renderer, Figure property 1-1676
RendererMode, Figure property 1-1680
repeatedly executing statements 1-1802
repeatedly executing statements in
  parallel 1-3829
replicating a matrix 1-4493
repmat 1-4493
resample (tscollection) 1-4495
reset 1-4498
reshape 1-4501
residue 1-4503
residues of transfer function 1-4503
Resize, Figure property 1-1681
ResizeFcn, Figure property 1-1681
restoredefaultpath 1-4507
rethrow 1-4508
rethrow, MException method 1-4510

return 1-4514
reverse
  array along dimension 1-1777
  array dimension 1-1777
  matrix column order 1-1778
  matrix row order 1-1779
reverse Cuthill-McKee ordering 1-5193 1-5203
RGB images
  converting to indexed 1-4517
RGB, converting to HSV 1-4516
rgb2hsv 1-4516
rgb2ind 1-4517
rgbplot 1-4521
ribbon 1-4523
right-click and context menus 1-5713
rmappdata function 1-4525
rmpref function 1-4534
rolling camera 1-617
root folder 1-3211
Root graphics object 1-4538
root object 1-4538
root, see rootobject 1-4538
roots 1-4552
roots of a polynomial 1-3991 to 1-3992 1-4552
rose 1-4554
Rosenbrock
  banana function 1-1790
  ODE solver 1-3656 1-3671 1-3686 1-3701
    1-3716 1-3731 1-3746
rosser 1-4557
rot90 1-4558
rotate 1-4559
rotate3d 1-4563
rotate3d mode objects 1-4563
rotating camera 1-609
rotating camera target 1-611
Rotation, Text property 1-5545
rotations
  Jacobi 1-4831
round 1-4569
  to nearest integer 1-4569
  towards infinity 1-856
  towards minus infinity 1-1781
  towards zero 1-1773
roundoff error
  characteristic polynomial and 1-3992
  effect on eigenvalues 1-388
  evaluating matrix functions 1-1880
  in inverse Hilbert matrix 1-2808
  partial fraction expansion and 1-4504
  polynomial roots and 1-4552
  sparse matrix conversion and 1-4798
RowName
  Uitable property 1-5902
RowStriping
  Uitable property 1-5903
rref 1-4570
rrefmovie 1-4570
rsf2csf 1-4572
rubberband box 1-4204
Runge-Kutta ODE solvers 1-3655 1-3670 1-3685
  1-3700 1-3715 1-3730 1-3745
running average 1-1732

S

save 1-4576 1-4583
  serial port I/O 1-4585
saveas 1-4587
savepath 1-4593
saving
  ASCII data 1-4576
  session to a file 1-1360
  workspace variables 1-4576
scalar product (of vectors) 1-1123
scattered data, aligning
  multi-dimensional 1-3454
scattergroup
  properties 1-4633

Schmidt semi-normalized Legendre functions 1-2992
schur 1-4652
Schur decomposition 1-4652
Schur form of matrix 1-4572 1-4652
ScreenDepth, Root property 1-4548
ScreenPixelsPerInch, Root property 1-4549
ScreenSize, Root property 1-4549
script 1-4654
search path
  MATLAB 1-3882
  modifying 1-3887
  toolbox folder 1-5643
  viewing 1-3887
search, string 1-1752
sec 1-4664
secant 1-4664
  hyperbolic 1-4668
  inverse 1-295
  inverse hyperbolic 1-298
sech 1-4668
Selected
  areaseries property 1-283
  Axes property 1-370
  barseries property 1-451
  contour property 1-1060
  errorbar property 1-1501
  Figure property 1-1683
  hggroup property 1-2508
  hgtransform property 1-2536
  Image property 1-2624
  Light property 1-3030
  Line property 1-3055
  lineseries property 1-3071
  Patch property 1-3878
  quivergroup property 1-4160
  rectangle property 1-4392
  Root property 1-4550
  scatter property 1-4646
  stairseries property 1-4891
  stem property 1-4960
  Surface property 1-5141
  surfaceplot property 1-5166
  Text property 1-5545
  Uicontrol property 1-5756
  Uitable property 1-5903
selecting areas 1-4204
SelectionHighlight
  areaseries property 1-284
  Axes property 1-371
  barseries property 1-451
  contour property 1-1061
  errorbar property 1-1502
  Figure property 1-1683
  hggroup property 1-2508
  hgtransform property 1-2536
  Image property 1-2625
  Light property 1-3030
  Line property 1-3055
  lineseries property 1-3072
  Patch property 1-3878
  quivergroup property 1-4161
  rectangle property 1-4392
  scatter property 1-4646
  stairseries property 1-4891
  stem property 1-4960
  Surface property 1-5141
  surfaceplot property 1-5166
  Text property 1-5545
  Uicontrol property 1-5756
  Uitable property 1-5903
SelectionType, Figure property 1-1683
selectmoveresize 1-4669
semicolon (special characters) 1-82
sendmail 1-4676
Separator
  Uipushtool property 1-5847
  Uitoggletool property 1-5917
Separator, Uimenu property 1-5802
serial 1-4679

`serialbreak` 1-4681
server variable 1-1610
session
  saving 1-1360
`set` 1-4682
  serial port I/O 1-4695
  timer object 1-4697
`set (tscollection)` 1-4700
`set` `hgsetget` class method 1-4691
`setabstime (tscollection)` 1-4701
`setappdata` 1-4702
`setDefaultStream`, RandStream method 1-4703
`setdiff` 1-4704
`setdisp` `hgsetget` class method 1-4714
`setenv` 1-4715
`setpixelposition` 1-4718
`setpref` function 1-4721
`setstr` 1-4723
`settimeseriesnames` 1-4727
`setxor` 1-4728
`shading` 1-4740
shading colors in surface plots 1-4740
`shiftdim` 1-4745
shifting array
  circular 1-915
`ShowArrowHead`
  quivergroup property 1-4161
`ShowBaseLine`
  barseries property 1-451
`ShowHiddenHandles`, Root property 1-4550
`showplottool` 1-4746
`ShowText`
  contour property 1-1061
`shrinkfaces` 1-4748
shutdown 1-4138
`sign` 1-4752
signum function 1-4752
simplex search 1-1792
Simpson's rule, adaptive recursive 1-4113
Simulink
  version number, comparing 1-6030
  version number, displaying 1-6022
sine
  hyperbolic 1-4757
  inverse hyperbolic 1-303
`single` 1-4756
single quote (special characters) 1-82
singular value
  decomposition 1-4200 1-5182
  rank and 1-4200
`sinh` 1-4757
`size`
  array dimesions 1-4758
  serial port I/O 1-4766
`size (tscollection)` 1-4768
size of array dimensions 1-4758
size of fonts, see also `FontSize` property 1-5548
size vector 1-4501
`SizeData`
  scatter property 1-4646
`SizeDataSource`
  scatter property 1-4647
`slice` 1-4769
slice planes, contouring 1-1069
sliders 1-5727
`SliderStep`, Uicontrol property 1-5756
smallest array elements 1-3344
`smooth3` 1-4775
smoothing 3-D data 1-4775
soccer ball (example) 1-5203
solution statistics (BVP) 1-592
`sort` 1-4780
sorting
  array elements 1-4780
  complex conjugate pairs 1-1112
  matrix rows 1-4784
`sortrows` 1-4784
sound
  files
    reading 1-328 1-6102

writing 1-330 1-6108
playing 1-318
recording 1-322
resampling 1-318
sampling 1-322
source control on UNIX platforms
checking out files
function 1-896
source control systems
checking in files 1-894
undo checkout 1-5934
`spalloc` 1-4793
`sparse` 1-4794
sparse matrix
allocating space for 1-4793
applying function only to nonzero elements of 1-4811
density of 1-3604
detecting 1-2910
diagonal 1-4799
finding indices of nonzero elements of 1-1736
identity 1-4810
number of nonzero elements in 1-3604
permuting columns of 1-1001
random 1-4829 to 1-4830
random symmetric 1-4831
replacing nonzero elements of with ones 1-4825
results of mixed operations on 1-4795
specifying maximum number of nonzero elements 1-4794
vector of nonzero elements 1-3606
visualizing sparsity pattern of 1-4842
sparse storage
criterion for using 1-1861
`spaugment` 1-4796
`spconvert` 1-4797
`spdiags` 1-4799
special characters
descriptions 1-2464
overloading 1-84
`specular` 1-4809
`SpecularColorReflectance`
Patch property 1-3878
Surface property 1-5142
surfaceplot property 1-5167
`SpecularExponent`
Patch property 1-3878
Surface property 1-5142
surfaceplot property 1-5167
`SpecularStrength`
Patch property 1-3878
Surface property 1-5142
surfaceplot property 1-5167
`speye` 1-4810
`spfun` 1-4811
`sph2cart` 1-4813
`sphere` 1-4814
sphereical coordinates
defining a Light position in 1-3032
spherical coordinates 1-4813
`spinmap` 1-4817
`spline` 1-4818
spline interpolation (cubic)
one-dimensional 1-2769 1-2777 1-2781 1-2785
Spline Toolbox 1-2773
`spones` 1-4825
`spparms` 1-4826
`sprand` 1-4829
`sprandn` 1-4830
`sprandsym` 1-4831
`sprank` 1-4833
spreadsheets
loading WK1 files 1-6160
loading XLS files 1-6212
reading into a matrix 1-1387
writing from matrix 1-6163
writing matrices into 1-1391
`sqrt` 1-4844

sqrtm 1-4845
square root
  of a matrix 1-4845
  of array elements 1-4844
  of real numbers 1-4370
squeeze 1-4848
stack, displaying 1-1229
standard deviation 1-4900
start
  timer object 1-4896
startat
  timer object 1-4897
startup 1-4899
startup file 1-4899
startup files 1-3209
State
  Uitoggletool property 1-5918
static text 1-5728
std 1-4900
step size (DDE)
  initial step size 1-1273
  upper bound 1-1273
step size (ODE) 1-1272 1-3755
  initial step size 1-3756
  upper bound 1-3756
stop
  timer object 1-4966
stopasync 1-4967
stopwatch timer 1-5598
storage
  allocated for nonzero entries (sparse) 1-3636
  sparse 1-4794
str2cell 1-879
str2double 1-4968
str2func 1-4969
str2mat 1-4973
str2num 1-4974
strcat 1-4978
stream lines
  computing 2-D 1-4988
  computing 3-D 1-4990
  drawing 1-4992
stream2 1-4988
stream3 1-4990
stretch-to-fill 1-339
strfind 1-5020
string
  converting from vector to 1-887
  converting matrix into 1-3196 1-3625
  converting to lowercase 1-3148
  converting to numeric array 1-4974
  converting to uppercase 1-5983
  dictionary sort of 1-4784
  finding first token in 1-5057
  searching and replacing 1-5047
  searching for 1-1752
String
  Text property 1-5546
  textarrow property 1-245
  textbox property 1-258
  Uicontrol property 1-5758
string matrix to cell array conversion 1-879
strings 1-5022
strjust 1-5027
strmatch 1-5028
strread 1-5038
strrep 1-5047
strtok 1-5057
strtrim 1-5061
struct2cell 1-5068
structure array
  getting contents of field of 1-1981
  setting contents of a field of 1-4716
structure arrays
  field names of 1-1640
structures
  dynamic fields 1-82
strvcat 1-5073
Style
  Light property 1-3030

Uicontrol property 1-5760
sub2ind 1-5075
subplot 1-5078
subplots
assymetrical 1-5083
suppressing ticks in 1-5085
subscripts
in text strings 1-5550
subspace 1-5092
subsref (function equivalent for A(i,j,k...)) 1-83
subtraction (arithmetic operator) 1-60
subvolume 1-5097
sum 1-5100
of array elements 1-5100
superscripts
in text strings 1-5550
support 1-5105
surf2patch 1-5114
surface 1-5116
Surface
and contour plotter 1-1586
converting to a patch 1-5114
creating 1-5116
defining default properties 1-4378 1-5120
plotting mathematical functions 1-1578
properties 1-5121 1-5145
surface normals, computing for volumes 1-2886
surfl 1-5173
svd 1-5182
svds 1-5184
swapbytes 1-5187
symamd 1-5192
symbfact 1-5196
symbols
operators 1-2464
symbols in text 1-246 1-258 1-5546
symmlq 1-5198
symrcm 1-5203
syntax, command 1-5206
syntax, function 1-5206
system folder
temporary 1-5219

T

table lookup. *See* interpolation
Tag
areaseries property 1-284
Axes property 1-371
barseries property 1-452
contour property 1-1061
errorbar property 1-1502
Figure property 1-1684
hggroup property 1-2508
hgtransform property 1-2536
Image property 1-2625
Light property 1-3030
Line property 1-3055
lineseries property 1-3072
Patch property 1-3879
quivergroup property 1-4161
rectangle property 1-4392
Root property 1-4550
scatter property 1-4647
stairseries property 1-4892
stem property 1-4960
Surface property 1-5142
surfaceplot property 1-5167
Text property 1-5551
Uicontextmenu property 1-5723
Uicontrol property 1-5760
Uimenu property 1-5802
Uipushtool property 1-5847
Uitable property 1-5903
Uitoggletool property 1-5918
Uitoolbar property 1-5929
Tagged Image File Format (TIFF)
writing 1-2673
tan 1-5213

tangent 1-5213
- four-quadrant, inverse 1-309
- hyperbolic 1-5216
- inverse 1-308
- inverse hyperbolic 1-315

`tanh` 1-5216
`tar` 1-5217
target, of camera 1-618
`tempdir` 1-5219
`tempname` 1-5220
temporary
- files 1-5220
- system folder 1-5219

tensor, Kronecker product 1-2952
terminating MATLAB 1-4138
test matrices 1-1900
test, logical. *See* logical tests *and* detecting
tetrahedron
- mesh plot 1-5221

`tetramesh` 1-5221
TeX commands in text 1-246 1-258 1-5546
`text` 1-5516
- editing 1-3967
- subscripts 1-5550
- superscripts 1-5550

Text
- creating 1-5516
- defining default properties 1-5520
- fixed-width font 1-5533
- properties 1-5521

`TextBackgroundColor`
- textarrow property 1-248

`TextColor`
- textarrow property 1-248

`TextEdgeColor`
- textarrow property 1-249

`TextLineWidth`
- textarrow property 1-249

`TextList`
- contour property 1-1062

`TextListMode`
- contour property 1-1062

`TextMargin`
- textarrow property 1-249

`textread` 1-5554
`TextRotation`, textarrow property 1-249
`TextStep`
- contour property 1-1063

`TextStepMode`
- contour property 1-1063

`textwrap` 1-5582
`tfqmr` 1-5585
`throw`, MException method 1-5591
`throwAsCaller`, MException method 1-5595
`TickDir`, Axes property 1-371
`TickDirMode`, Axes property 1-372
`TickLength`, Axes property 1-372
TIFF
- compression 1-2682
- encoding 1-2677
- `ImageDescription` field 1-2682
- maxvalue 1-2677
- parameters that can be set when writing 1-2681
- resolution 1-2682
- writemode 1-2682
- writing 1-2673

TIFF image format
- specifying color space 1-2681

tiling (copies of a matrix) 1-4493
time
- CPU 1-1113
- elapsed (stopwatch timer) 1-5598

time and date functions 1-1465
timer
- properties 1-5618
- timer object 1-5618

`timerfind`
- timer object 1-5626

`timerfindall`

timer object 1-5628
times (function equivalent for .*) 1-65
timeseries
getdatasamples method 1-4259
timeseries object
methods 1-4235
timestamp 1-1365
Title, Axes property 1-372
todatenum 1-5641
toeplitz 1-5642
Toeplitz matrix 1-5642
toggle buttons 1-5728
token 1-5057
See also string
Toolbar
Figure property 1-1685
Toolbox
Spline 1-2773
toolbox folder, path 1-5643
toolboxdir 1-5643
TooltipString
Uicontrol property 1-5760
Uipushtool property 1-5848
Uitable property 1-5904
Uitoggletool property 1-5918
trace 1-5645
trace of a matrix 1-1356 1-5645
trailing blanks
removing 1-1280
transform
hgtransform function 1-2516
transform, Fourier
discrete, n-dimensional 1-1622
discrete, one-dimensional 1-1615
discrete, two-dimensional 1-1620
inverse, n-dimensional 1-2592
inverse, one-dimensional 1-2588
inverse, two-dimensional 1-2590
shifting the zero-frequency component of 1-1625
transformation
See also conversion 1-653
transpose
array (arithmetic operator) 1-62
matrix (arithmetic operator) 1-62
transpose (function equivalent for .\q) 1-66
trapz 1-5647
treelayout 1-5649
treeplot 1-5650
triangulation
2-D plot 1-5657
tril 1-5652
trimesh 1-5653
triple integral
numerical evaluation 1-5655
triplequad 1-5655
triplot 1-5657
trisurf 1-5671
triu 1-5673
true 1-5674
truth tables (for logical operations) 1-71
tscollection 1-5678
tsdata.event 1-5681
tsearchn 1-5682
tstool 1-5683
type 1-5684
Type
areaseries property 1-284
Axes property 1-373
barseries property 1-452
contour property 1-1063
errorbar property 1-1502
Figure property 1-1685
hggroup property 1-2509
hgtransform property 1-2537
Image property 1-2625
Light property 1-3031
Line property 1-3055
lineseries property 1-3072
Patch property 1-3879

quivergroup property 1-4162
rectangle property 1-4393
Root property 1-4550
scatter property 1-4648
stairseries property 1-4892
stem property 1-4961
Surface property 1-5142
surfaceplot property 1-5168
Text property 1-5551
Uicontextmenu property 1-5724
Uicontrol property 1-5761
Uimenu property 1-5802
Uipushtool property 1-5848
Uitable property 1-5904
Uitoggletool property 1-5919
Uitoolbar property 1-5929
`typecast` 1-5685

U

`UData`
errorbar property 1-1503
quivergroup property 1-4163
`UDataSource`
errorbar property 1-1503
quivergroup property 1-4163
Uibuttongroup
defining default properties 1-5694
`uibuttongroup` function 1-5689
Uibuttongroup Properties 1-5694
`uicontextmenu` 1-5713
`UiContextMenu`
Uicontrol property 1-5761
Uipushtool property 1-5848
Uitoggletool property 1-5919
Uitoolbar property 1-5930
`UIContextMenu`
areaseries property 1-285
Axes property 1-373
barseries property 1-452
contour property 1-1063
errorbar property 1-1503
Figure property 1-1685
hggroup property 1-2509
hgtransform property 1-2537
Image property 1-2626
Light property 1-3031
Line property 1-3056
lineseries property 1-3072
Patch property 1-3880
quivergroup property 1-4162
rectangle property 1-4393
scatter property 1-4648
stairseries property 1-4892
stem property 1-4961
Surface property 1-5143
surfaceplot property 1-5168
Text property 1-5551
Uitable property 1-5904
Uicontextmenu Properties 1-5716
`uicontrol` 1-5725
Uicontrol
defining default properties 1-5736
fixed-width font 1-5746
types of 1-5725
Uicontrol Properties 1-5736
uicontrols
printing 1-4032
`uigetdir` 1-5764
`uigetfile` 1-5768
`uigetpref` function 1-5779
`uiimport` 1-5786
`uimenu` 1-5787
Uimenu
creating 1-5787
defining default properties 1-5791
Properties 1-5791
Uimenu Properties 1-5791
Uipanel
defining default properties 1-5818

uipanel function 1-5815
Uipanel Properties 1-5818
uipushtool 1-5836
Uipushtool
  defining default properties 1-5839
Uipushtool Properties 1-5839
uiputfile 1-5850
uiresume 1-5859
uisetcolor function 1-5863
uisetfont 1-5864
uisetpref function 1-5866
uistack 1-5867
Uitable
  defining default properties 1-5876
  fixed-width font 1-5895
uitable function 1-5868
Uitable Properties 1-5876
uitoggletool 1-5906
Uitoggletool
  defining default properties 1-5908
Uitoggletool Properties 1-5908
uitoolbar 1-5920
Uitoolbar
  defining default properties 1-5922
Uitoolbar Properties 1-5922
uiwait 1-5931
uminus (function equivalent for unary \xd0
) 1-65
unconstrained minimization 1-1788
undefined numerical results 1-3435
undocheckout 1-5934
unicode2native 1-5935
union 1-5936
unique 1-5948
Units
  annotation ellipse property 1-233
  annotation line property 1-236
  annotation rectangle property 1-240
  arrow property 1-225
  Axes property 1-373
  doublearrow property 1-230
  Figure property 1-1686
  Root property 1-4551
  Text property 1-5551
  textarrow property 1-249
  textbox property 1-261
  Uicontrol property 1-5761
  Uitable property 1-5904
unlocking functions 1-3432
unmkpp 1-5966
untar 1-5973
unwrap 1-5975
unzip 1-5980
up vector, of camera 1-620
updating figure during file execution 1-1412
uplus (function equivalent for unary +) 1-65
upper 1-5983
upper triangular matrix 1-5673
uppercase to lowercase 1-3148
user input
  from a button menu 1-3260
UserData
  areaseries property 1-285
  Axes property 1-374
  barseries property 1-453
  contour property 1-1064
  errorbar property 1-1504
  Figure property 1-1686
  hggroup property 1-2509
  hgtransform property 1-2538
  Image property 1-2626
  Light property 1-3031
  Line property 1-3056
  lineseries property 1-3073
  Patch property 1-3880
  quivergroup property 1-4162
  rectangle property 1-4393
  Root property 1-4551
  scatter property 1-4648
  stairseries property 1-4893

stem property 1-4961
Surface property 1-5143
surfaceplot property 1-5168
Text property 1-5552
Uicontextmenu property 1-5724
Uicontrol property 1-5762
Uimenu property 1-5802
Uipushtool property 1-5849
Uitable property 1-5905
Uitoggletool property 1-5919
Uitoolbar property 1-5930

V

Value, Uicontrol property 1-5762
vander 1-6014
Vandermonde matrix 1-4001
var 1-6015
varargin 1-6017
varargout 1-6019
variable numbers of arguments 1-6019
variable-order solver (ODE) 1-3764
variables
checking existence of 1-1532
global 1-2021
in workspace 1-6165
keeping some when clearing 1-943
linking to graphs with linkdata 1-3091
listing 1-6139
local 1-2021
name of passed 1-2715
persistent 1-3936
saving 1-4576
sizes of 1-6139
VData
quivergroup property 1-4164
VDataSource
quivergroup property 1-4164
vector
dot product 1-1408
frequency 1-3145
product (cross) 1-1123
vector field, plotting 1-1023
vectorize 1-6021
vectorizing ODE function (BVP) 1-589
vectors, creating
logarithmically spaced 1-3145
regularly spaced 1-85 1-3108
velocity vectors, plotting 1-1023
verctrl function (Windows) 1-6026
verLessThan 1-6030
version 1-6032
version numbers
comparing 1-6030
displaying 1-6022
vertcat 1-6034
vertcat (function equivalent for [1-83
vertcat (tscollection) 1-6036
VertexNormals
Patch property 1-3880
Surface property 1-5143
surfaceplot property 1-5168
VerticalAlignment, Text property 1-5552
VerticalAlignment, textbox property 1-250 1-261
Vertices, Patch property 1-3880
video
saving in AVI format 1-331
view 1-6054
azimuth of viewpoint 1-6054
coordinate system defining 1-6054
elevation of viewpoint 1-6054
view angle, of camera 1-622
View, Axes property (obsolete) 1-374
viewing
a group of object 1-607
a specific object in a scene 1-607
viewmtx 1-6057
Visible
areaseries property 1-285

Axes property 1-375
barseries property 1-453
contour property 1-1064
errorbar property 1-1504
Figure property 1-1687
hggroup property 1-2509
hgtransform property 1-2538
Image property 1-2626
Light property 1-3031
Line property 1-3056
lineseries property 1-3073
Patch property 1-3880
quivergroup property 1-4162
rectangle property 1-4393
Root property 1-4551
scatter property 1-4648
stairseries property 1-4893
stem property 1-4962
Surface property 1-5143
surfaceplot property 1-5169
Text property 1-5553
Uicontextmenu property 1-5724
Uicontrol property 1-5763
Uimenu property 1-5802
Uipushtool property 1-5849
Uitable property 1-5905
Uitoggletool property 1-5919
Uitoolbar property 1-5930
visualizing
cell array structure 1-877
sparse matrices 1-4842
volumes
calculating isosurface data 1-2889
computing 2-D stream lines 1-4988
computing 3-D stream lines 1-4990
computing isosurface normals 1-2886
contouring slice planes 1-1069
drawing stream lines 1-4992
end caps 1-2879
reducing face size in isosurfaces 1-4748
reducing number of elements in 1-4401
voronoi 1-6068
Voronoi diagrams
multidimensional vizualization 1-6075
two-dimensional vizualization 1-6068
voronoin 1-6075

W

wait
timer object 1-6079
waitbar 1-6080
waitfor 1-6084
waitforbuttonpress 1-6088
warndlg 1-6090
warning 1-6093
warning message (enabling, suppressing, and displaying) 1-6093
waterfall 1-6096
.wav files
reading 1-6102
writing 1-6108
wavfinfo 1-6099
wavread 1-6099 1-6102
wavwrite 1-6108
WData
quivergroup property 1-4164
WDataSource
quivergroup property 1-4165
well conditioned 1-4206
what 1-6122
whatsnew 1-6126
white space characters, ASCII 1-2909 1-5057
whitebg 1-6137
wilkinson 1-6154
Wilkinson matrix 1-4803 1-6154
WindowButtonDownFcn, Figure property 1-1687
WindowButtonMotionFcn, Figure property 1-1688
WindowButtonUpFcn, Figure property 1-1689

WindowKeyPressFcn, Figure property 1-1689
WindowKeyReleaseFcn , Figure property 1-1691
Windows Paintbrush files
  writing 1-2672
WindowScrollWheelFcn, Figure property 1-1692
WindowStyle, Figure property 1-1695
winopen 1-6155
winqueryreg 1-6157
WK1 files
  loading 1-6160
  writing from matrix 1-6163
wk1finfo 1-6159
wk1read 1-6160
wk1write 1-6163
workspace 1-6165
  changing context while debugging 1-1223
    1-1246
  consolidating memory 1-3813
  predefining variables 1-4899
  saving 1-4576
  variables in 1-6139
  viewing contents of 1-6165
workspace variables
  reading from disk 1-3116
WVisual, Figure property 1-1697
WVisualMode, Figure property 1-1699

X

X
  annotation arrow property 1-226 1-231
  annotation line property 1-237
  textarrow property 1-250
X Windows Dump files
  writing 1-2673
x-axis limits, setting and querying 1-6198 1-6202
  1-6206
XAxisLocation, Axes property 1-375
XColor, Axes property 1-375
XData
  areaseries property 1-285
  barseries property 1-453
  contour property 1-1064
  errorbar property 1-1504
  Image property 1-2626
  Line property 1-3056
  lineseries property 1-3073
  Patch property 1-3881
  quivergroup property 1-4165
  scatter property 1-4649
  stairseries property 1-4893
  stem property 1-4962
  Surface property 1-5143
  surfaceplot property 1-5169
XDataMode
  areaseries property 1-286
  barseries property 1-453
  contour property 1-1064
  errorbar property 1-1504
  lineseries property 1-3073
  quivergroup property 1-4166
  stairseries property 1-4893
  stem property 1-4962
  surfaceplot property 1-5169
XDataSource
  areaseries property 1-286
  barseries property 1-454
  contour property 1-1065
  errorbar property 1-1505
  lineseries property 1-3074
  quivergroup property 1-4166
  scatter property 1-4649
  stairseries property 1-4894
  stem property 1-4962
  surfaceplot property 1-5169
XDir, Axes property 1-376
XDisplay, Figure property 1-1699
XGrid, Axes property 1-376
xlabel 1-6184
XLabel, Axes property 1-377

xlim 1-6198 1-6202 1-6206
XLim, Axes property 1-377
XLimMode, Axes property 1-378
XLS files
    loading 1-6212
xlsfinfo 1-6210
XMinorGrid, Axes property 1-378
xor 1-6233
XScale, Axes property 1-379
xslt 1-6234
XTick, Axes property 1-379
XTickLabel, Axes property 1-379
XTickLabelMode, Axes property 1-380
XTickMode, Axes property 1-380
XVisual, Figure property 1-1700
XVisualMode, Figure property 1-1702
XWD files
    writing 1-2673
xyz coordinates . *See* Cartesian coordinates

Y

Y
    annotation arrow property 1-226 1-231 1-237
    textarrow property 1-250
y-axis limits, setting and querying 1-6198 1-6202 1-6206
YAxisLocation, Axes property 1-375
YColor, Axes property 1-375
YData
    areaseries property 1-287
    barseries property 1-454
    contour property 1-1065
    errorbar property 1-1505
    Image property 1-2627
    Line property 1-3057
    lineseries property 1-3074
    Patch property 1-3881
    quivergroup property 1-4167
    scatter property 1-4650
    stairseries property 1-4894
    stem property 1-4963
    Surface property 1-5144
    surfaceplot property 1-5170
YDataMode
    contour property 1-1066
    quivergroup property 1-4167
    surfaceplot property 1-5170
YDataSource
    areaseries property 1-287
    barseries property 1-455
    contour property 1-1066
    errorbar property 1-1506
    lineseries property 1-3075
    quivergroup property 1-4167
    scatter property 1-4650
    stairseries property 1-4895
    stem property 1-4963
    surfaceplot property 1-5170
YDir, Axes property 1-376
YGrid, Axes property 1-376
ylabel 1-6184
YLabel, Axes property 1-377
ylim 1-6198 1-6202 1-6206
YLim, Axes property 1-377
YLimMode, Axes property 1-378
YMinorGrid, Axes property 1-378
YScale, Axes property 1-379
YTick, Axes property 1-379
YTickLabel, Axes property 1-379
YTickLabelMode, Axes property 1-380
YTickMode, Axes property 1-380

Z

z-axis limits, setting and querying 1-6198 1-6202 1-6206
ZColor, Axes property 1-375
ZData
    contour property 1-1067

Line property 1-3057
lineseries property 1-3075
Patch property 1-3881
quivergroup property 1-4168
scatter property 1-4650
stemseries property 1-4964
Surface property 1-5144
surfaceplot property 1-5171
`ZDataSource`
contour property 1-1067
lineseries property 1-3075 1-4964
scatter property 1-4651
surfaceplot property 1-5171
`ZDir`, Axes property 1-376
`ZGrid`, Axes property 1-376
Ziggurat 1-4188 1-4195
zlabel 1-6184
zlim 1-6198 1-6202 1-6206
`ZLim`, Axes property 1-377
`ZLimMode`, Axes property 1-378
`ZMinorGrid`, Axes property 1-378
`zoom` 1-6246
zoom mode objects 1-6247
`ZScale`, Axes property 1-379
`ZTick`, Axes property 1-379
`ZTickLabel`, Axes property 1-379
`ZTickLabelMode`, Axes property 1-380
`ZTickMode`, Axes property 1-380